Software Platforms for Modern Embedded Systems

Techniques Towards Productivity, Reliability and Scalability

Yixiao Li

Abstract

The increasing complexity of modern embedded systems makes software platform techniques more necessary than ever. Even the cheapest devices can have advanced connectivity now. For high-end embedded systems (e.g. autonomous vehicles), multi/many-core processors are adopted to satisfy the growing demand for computing power. Productivity, reliability and scalability are essential requirements for a software platform. In this dissertation, some major open issues are introduced at first. Three studies – EV3RT, FMP-MC, ESPROF – are then presented to shed some lights on and discuss possible solutions to them.

In the study of EV3RT, how to build a reliable RTOS-based platform meeting both real-time performance and connectivity requirements, with significantly reduced implementation effort, for Mindstorms EV3 robotics kit is explained. A dynamic module loading mechanism for static OS design is proposed to improve the productivity of development process. The performance and footprint are compared with Linux-based platforms to show the advantages for resource- and time-critical applications.

In the study of FMP-MC, a testbed for running high-performance applications on traditional multi-core RTOS is created. By a comparative analysis with Linux on a many-core processor, several bottlenecks commonly existing in RTOSes are identified and resolved. Multiple parallel applications from PARSEC are used for evaluation. The results indicate that traditional multi-core RTOS can be optimized to deliver good scalability on many-core.

In the study of ESPROF, a generic source-level profiling infrastructure for multi/many-core embedded systems is proposed. It allows user to flexibly and effortlessly create optimized tools with advanced algorithms. A scalable call graph profiler is implemented as an example, and shows much higher accuracy with very low overhead compared to existing tool in measuring benchmark application on a 36-core platform.

Besides proposed solutions to those directly addressed issues, all the studies are open-source and can provide some conceptual and practical bases for further research. The state of the art and some noteworthy trends about techniques related to software platforms for future embedded systems are also discussed.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Hiroaki Takada, for giving me the opportunity to study in his laboratory. I also thank my advisor, Prof. Yutaka Matsubara. I would not have been able to complete studies in this dissertation without their invaluable guidance, support and advice throughout my PhD.

Special thanks to all those people who have offered help in my life, especially during hard times such as the recovery period after a surgery.

I owe a debt of gratitude to the open-source-software movement. Open-source projects formed the basis of my research. Some (e.g. Visual Studio Code from Microsoft and Markdown Monster from West Wind Technologies) also boosted my daily productivity.

Financial support from the NGK SPARK PLUG foreign student scholarship is gratefully acknowledged.

Last but not least, I thank my family and friends for the constant support, encouragement and understanding.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

ABI Application Binary Interface

API Application Programming Interface

AST Abstract Syntax Tree

AUTOSAR AUTomotive Open System ARchitecture

CCT Calling Context Tree

CFS Completely Fair Scheduler

CSL Core Services Layer

DDC Dynamic Distributed Cache

FMP Flexible Multiprocessor Profile

GPOS General-Purpose Operating System

GUI Graphical User Interface

HPC High Performance Computing

IDE Integrated Development Environment

IMA Integrated Modular Avionics

IoT Internet of Things

PIL Platform Interface Layer

PWM Pulse Width Modulation

RTE RunTime Environment

RTOS Real-Time Operating System

SDK Software Development Kit

SLOC Source Lines Of Code

SoC System On a Chip

SSH Secure SHell

TCB Trusted Computing Base

TEE Trusted Execution Environment

TLSF Two-Level Segregated Fit

UAL User Application Layer

Chapter 1

Introduction

1.1 Background

An embedded system is a dedicated computer system to perform specific functions. Typical examples of applications include consumer electronics, vehicle control and factory automation. Many tasks in these systems interact with the physical environment and thus have real-time constraints. For embedded systems targeting mass production, the manufacturing cost of hardware is a crucial concern. Energy efficiency is also important because they often run on batteries. Therefore, software implementations in embedded systems are highly optimized for low overhead and small footprint to use a minimum amount of hardware resources.

The size and complexity of software in embedded systems can vary a lot for different application domains. Applications for low cost embedded devices are usually built on a lightweight RTOS (Real-Time Operating System) kernel (e.g. FreeRTOS[4], TOPPERS[131]). Many extremely resource-constrained devices even use bare-metal applications, which directly run on the hardware without any underlying kernel, to eliminate the footprint of RTOS kernel [33]. On the other hand, in industries like avionics and automotive, software can be very large in scale. For example, a Boeing 747 airliner has tens of millions of SLOC, and system for a recent car can even exceed 100 million SLOC [94].

For large-scale embedded systems, platform-based design [110] has been applied for years as a powerful methodology to cope with the ever-increasing pressure on development cost and time-to-market. Famous examples include IMA [47] for avionics and AUTOSAR [27] for automotive. While the definition of "software platform" is very loose and domain-dependent, a generalized overview is shown in Fig. 1.1.

The development of applications is based on a framework which usually includes domain-specific APIs, management system for software modules/components and common toolsets

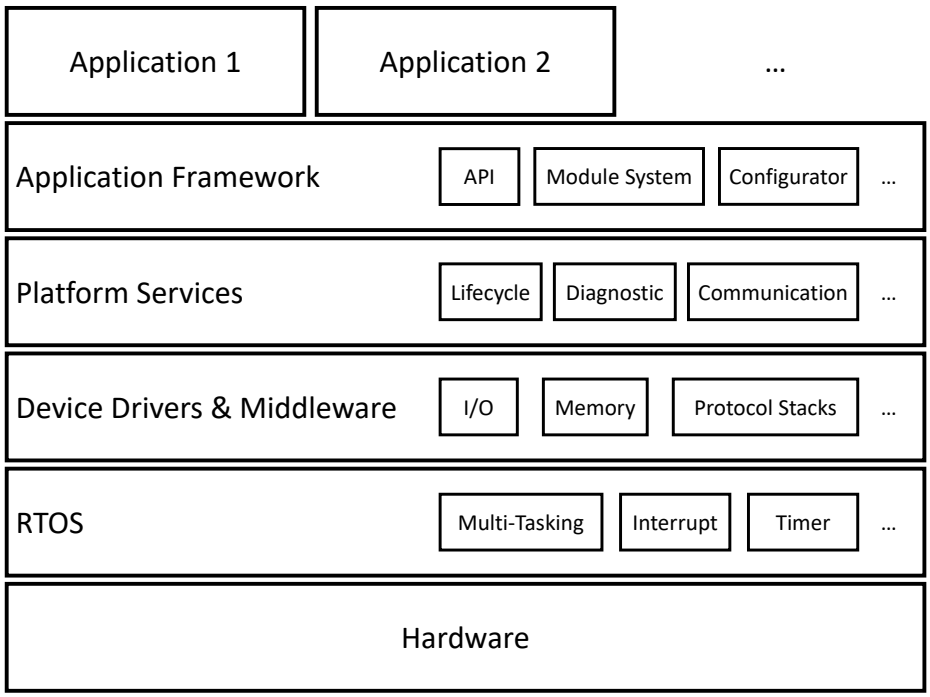

Fig. 1.1 Generalized architecture of layered software platform

(e.g. code configurator, compiler, analyzer). A module system specifies the boundary of an application to build and deploy. It may also allow developers to define interfaces between software packages to support component-based development [13]. In most cases, application modules are statically linked with the platform as a monolithic image. The framework should have stable and versioned specifications to enable the maintainability and reusability of applications.

Device drivers and middleware provide interfaces to control the hardware. They should be implemented upon the RTOS in order to cooperate with other components using the kernel services. Low-level hardware abstractions (e.g. interrupt controller and timer) are considered as a part of the RTOS rather than device drivers in this figure.

Platform must provide necessary services supporting the framework functionalities. For instance, it receives (and checks) a domain-specific API invocation and then uses device drivers to accomplish the request. It manages the resources allocated to each application, and decides the lifecycle of (i.e. how and when to start/terminate) an application. If multiple applications can run at the same time, it should also provide a mechanism for communication. Over-the-Air updating, diagnostics and network manager are some examples of other possible platform services.

Modern embedded systems can accomplish much more complex tasks than before, benefiting from rapid advances in technology. Ubiquitous connectivity and multi/many-

core architecture have gained significant importance in the industry. The hardware cost of supporting connectivity has drastically dropped (e.g. $2 SoC with WiFi [112]). It makes the IoT (Internet of Things) concept, which delivers better/smarter services by connecting more physical objects to the internet, feasible. Further, for high-end embedded systems (e.g. autonomous vehicles), multi/many-core processors are adopted to satisfy the increasing performance requirements. Besides traditional real-time applications, these systems also tend to include high-performance parallel applications.

The increasing complexity of embedded systems makes software platform techniques more necessary than ever. Meanwhile, it brings many new challenges and questions, especially in terms of productivity, reliability and scalability. Some of the major ones are listed as follows.

(1) **Productivity:** Many recent mid-range and high-end devices are capable of running Linux-based platforms out-of-the-box. What are the costs and benefits to build and use RTOS-based platforms? Are they still appealing to those hardware systems?

(2) **Reliability:** Existing IoT-oriented platforms mainly focus on extending connectivity. For applications like industrial IoT, the reliability is as much as, if not more, important. How to enhance the reliability of a platform?

(3) **Productivity, Reliability:** Software updating has become a key feature for connected devices to remove bugs and improve functionality. For some applications, such as safety-critical ones, system reboots can be expensive, and thus dynamic updating is desirable. Meanwhile, in existing systems, static design (e.g. allocating kernel objects at compile time) is a popular approach to reduce resource usage and improve real-time performance. Can updating in a static OS-based platform be less disruptive?

(4) **Scalability:** Most current embedded multi-core processors only have several cores. Therefore, traditional platforms are not designed to run high-performance parallel applications. Potential bottlenecks may be triggered and cause poor performance as the number of cores increases. In order to deliver good scalability, it is vital to identify and deal with those problems.

(5) **Productivity, Scalability:** Performance analysis tools are essential for finding application-internal bottlenecks in parallel computing. Existing tools for HPC-oriented platforms are not suitable for embedded systems, especially in terms of resource usage and overhead. Meanwhile, creating new analysis tools is not easy due to the diversity in hardware and software architectures.

1.2 Overview

This dissertation consists of three contributions to shed some lights on and discuss possible solutions to those open issues.

In Chapter 2, issues (1) (2) (3) are addressed in a specific, yet representative, context: software platforms for LEGO Mindstorms EV3 robotics kit. EV3 has significant improvements in performance and connectivity compared to its predecessor (Mindstorms NXT), and is the first model supporting Linux-based platforms. While RTOS-based platforms are usually advantageous for real-time applications, creating a new platform is difficult and costly due to the increased complexity. No RTOS for NXT has supported protection functionalities, but the reliability of RTOS for EV3 must be enhanced. Otherwise, the software scale growth will make the platform much more susceptible to failure. In traditional development process, the platform and application are statically linked as a monolithic image, and thus updating requires reboot and drops all connections. Meanwhile, robot control applications, especially to users attending competition events, will be updated very frequently during development. Therefore, dynamic updating is an essential requirement for the productivity.

EV3RT, the first RTOS-based software platform for LEGO Mindstorms EV3 robotics kit, is proposed. It has a representative architecture for modern systems with both real-time performance and connectivity requirements. Policies, built upon its static kernel with protection functionalities, are applied to improve reliability. A dynamic module loading mechanism for static OS design is proposed to support rebootless application updating. By techniques like reusing device drivers from stock firmware, the implementation cost has been greatly saved. Performance evaluation results show that EV3RT can deliver highly predictable performance with very small overhead and footprint compared to Linux-based platforms.

In Chapter 3, issue (4) is addressed in an empirical way. A testbed allowing existing RTOS to run parallel applications on many-core processor is highly desirable and must be created at first. Comparing the testbed with Linux-based platform optimized for HPC is very likely to give us valuable hints about scalability problems in RTOS. To evaluate the effectiveness of our solutions, multiple real applications, in addition to microbenchmarks, should be used.

FMP-MC, a prototype platform for many-core embedded systems, is proposed. It is based on a traditional multi-core RTOS but includes necessary runtime system for running high-performance applications on a 72-core processor. Several bottlenecks in it are identified and methods to avoid them are proposed, through a comparative analysis between FMP-MC and Linux-based runtime system. PARSEC, a popular benchmark suite composed of

parallel applications, is used to evaluate the performance. The results show that, after proper optimization, FMP-MC can deliver better scalability than Linux in many cases.

In Chapter 4, a preliminary study towards issue (5) is described. Since traditional RTOS-based platforms are not for parallel applications, performance analysis tools barely exist. We have to begin by reviewing conventional methods and conducting requirements analysis. Some common obstacles before further research should be addressed. It is desirable to implement a proof of concept to evaluate our proposal.

ESPROF, a generic source-level profiling infrastructure, is proposed. In the HPC domain, large monolithic profiling tools are preferred. For multi/many-core embedded systems, after discussing their characteristics and existing techniques, we believe that multiple small tools are much more suitable. ESPROF allows users to effortlessly create new tools and also provides flexibility for optimization. A scalable and optimized call graph profiler is developed as an example. It shows much higher accuracy with very low overhead compared to existing tool, in the evaluation of measuring parallel application.

Finally, in Chapter 5, we conclude this dissertation and discuss the state of the art and some noteworthy trends about techniques related to software platforms for future embedded systems.

Chapter 2

EV3RT: Real-time Software Platform for LEGO Mindstorms EV3

2.1 Introduction

Mindstorms [73] has become one of the most popular series of robotics development kits since it was first released by LEGO Inc. in 1998. A Mindstorms kit consists of a programmable brick computer called intelligent brick that controls the whole system, and a set of modular sensors, motors and LEGO blocks that allow users to build robots flexibly. Many real-life embedded systems can be modelled with Mindstorms robots, and thus they are used as important tools in researches [71, 17] and college education like real-time control and artificial intelligence [26, 64]. Robotics competitions such as World Robot Olympiad [140] and RoboCup Junior [124] also utilize the Mindstorms robots.

LEGO Mindstorms EV3 (or just the EV3) [135] is the third and the latest (as of writing) generation of Mindstorms series. It supports many new features and is much more powerful than its predecessor, the LEGO Mindstorms NXT [45] series, as shown in Table 2.1.

The standard software platform of EV3, however, has some disadvantages when developing applications with real-time requirements. It consists of an integrated development environment (IDE) and a Linux-based firmware. The IDE is based on LabVIEW [90] which uses a visual programming language as shown in Fig. 2.1. It is proprietary software with very limited extensibility. While it is friendly to beginners of computer programming, users who are already familiar with common programming languages like C or C++ may find it difficult to develop complex programs in this IDE. Multitasking features such as preemptive scheduling and synchronization are also unsupported. The Linux-based firmware (codename: lms2012) includes a graphical user interface (GUI) program to operate the intelligent brick

Table 2.1 Comparison of Mindstorms NXT and Mindstorms EV3

	Mindstorms NXT	Mindstorms EV3
Processor	ARM7@48MHz	ARM9@300MHz
Coprocessor	Atmel AVR@8MHz	n/a
ROM	256KB	16MB
RAM	64KB	64MB
Sensors	Analog, I2C	Analog, I2C, UART
Motors	Up to 3	Up to 4
Display	100×64 LCD	178×128 LCD
SD card	n/a	microSD (SDHC)
Bluetooth	v1.2	v2.1 + EDR
USB device	12Mbps	480Mbps
USB host	n/a	12Mbps
WiFi	n/a	USB dongle

and a virtual machine-based runtime, which takes a long time to boot up and has a huge memory footprint. Application will be compiled into an intermediate representation (a.k.a bytecode) to be executed in the virtual machine. The overhead and unpredictable performance of Linux kernel and virtual machine can not provide real-time guarantees. There are some other software platforms such as leJOS EV3 [123] and MonoBrick [120], which support developing in Java or C#. However, they also use Linux and have their own runtime, and thus the disadvantages of lms2012 still remain.

In this chapter, EV3RT, a real-time software platform to overcome above disadvantages, is presented. It is the first RTOS-based platform for EV3 to our knowledge and especially suitable for developing applications with hard real-time requirements such as balancing a two-wheeled robot [42]. Soft real-time and generic applications, including those for IoT devices or just entertainment, are also likely to run faster and smoother on EV3RT with its low-overhead APIs. ET Robocon (short for Embedded Technology Software Design Robot Contest) [60], a popular robot competition in Japan, has selected EV3RT as one of its officially supported platforms. Many participating teams of ET Robocon are using EV3RT to make their robots accomplish the assigned tasks more stably and precisely. Main features of EV3RT are listed as follows:

TOPPERS/HRP2 static RTOS kernel[129]. As an extended version of TOPPERS/JSP kernel, which is used by a popular RTOS-based platform for NXT called nxtOSEK/JSP [31], it can provide backward compatibility as long as many new features with guaranteed real-time performance.

Programming in C and C++. Applications can be compiled and run as native code so the overhead of virtual machine is completely eliminated.

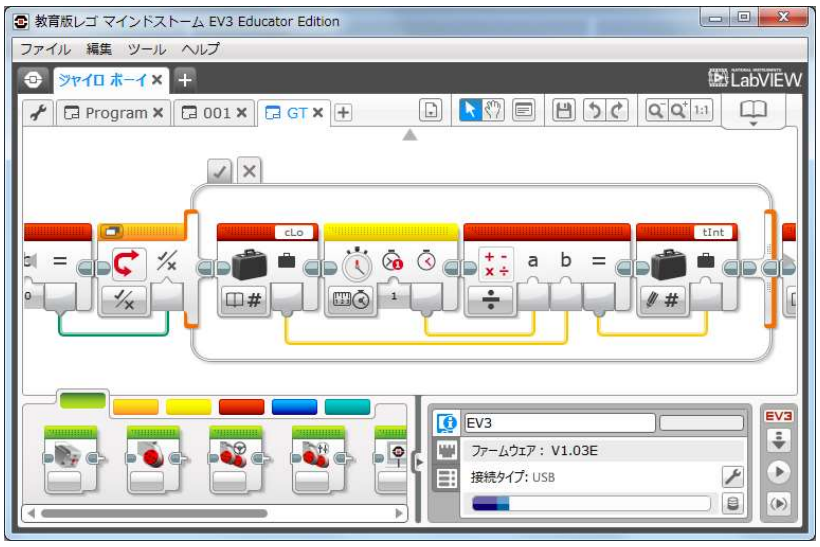

Fig. 2.1 Visual programming in LabVIEW

High reliability with enhanced protection. The base system of EV3RT is protected so it will not be affected by defects in user applications, which makes the debugging effortless for a large-scale embedded software platform like EV3RT.

Easy-to-use and low-overhead APIs. Users can easily develop applications with real-time requirement and need not learn the complex implementation details of various device drivers.

Dynamic application loading. A static RTOS generally requires the system to be rebooted when the application is updated. In the case of EV3, developers have to reset the intelligent brick and reconnect network such as Bluetooth every time after modifying an application, which will waste lots of time, especially when developing robot control applications which are usually updated frequently to adjust control logic and parameters. To solve this problem, EV3RT provides a loader for the static TOPPERS/HRP2 kernel, which allows the dynamic updating of application without reboot.

Bluetooth and USB device support. Connectivity technologies are important in many scenarios, such as creating interactive applications or modeling an IoT device. Helpful features like uploading application wirelessly are also supported.

Ultra-fast boot process. The slow startup is one of the most annoying problems for EV3 developers using Linux-based platforms. Meanwhile, it only takes about 2 seconds for EV3RT to boot up.

Very small memory footprint. Storing or caching data in memory as possible is a common technique to improve performance and reduce latency. More than 95% of total memory can be used by applications on EV3RT.

Complete open source platform[44, 43]. Since Mindstorms robots have been used in so many different ways, the standard features and behaviors of EV3RT may not fit some users' needs. In that case, users can freely modify any part of EV3RT as all the source code is available.

Extendable software architecture. The hardware of EV3 itself is extendable by connecting devices like third party sensors. Supporting of those devices can be easily implemented with the flexible architecture provided by EV3RT.

The rest of this chapter is organized as follows. An overview of EV3RT is given in Section 2.2 at first, by describing its architecture and usage. In Section 2.3, TOPPERS/HRP2 kernel, the RTOS of EV3RT, is explained, focusing on applying its protection functionalities to make EV3RT more reliable. A dynamic module loading mechanism for static OS design is proposed in Section 2.4, in order to implement the application loader for EV3RT. We then introduce some implementation techniques such as reusing Linux device drivers in Section 2.6. The performance of EV3RT is evaluated and compared with existing software platforms in Section 2.7. Finally, the chapter is concluded in Section 2.8.

2.2 Overview of EV3RT

EV3RT is a developer-friendly open source software platform for real-time applications. In this section, we first give an overview of EV3RT through explaining its architecture. Thereafter, how to develop application with EV3RT in practice is also briefly described.

2.2.1 Platform Architecture

Application in EV3RT can be developed and built in two modes: dynamic loading mode and standalone mode. In dynamic loading mode, a base system with application loader is running on the intelligent brick. Application is separately built as a module (in ELF file format [87]) which can be dynamically loaded from micro SD card or Bluetooth and then executed under non-privileged mode on the base system. On the contrary, application in standalone mode will be compiled and linked together with the whole platform, to generate a boot image supported by the EV3's bootloader U-Boot [38]. Both modes have their own merits. Application in dynamic loading mode can be updated without rebooting EV3, which makes the development process very smooth. Its building is also much faster than in standalone mode since compilation of the whole platform is avoided. Robot control applications are usually updated frequently to adjust control logic and parameters, and thus we highly recommended dynamic loading mode to be used. On the other hand, application

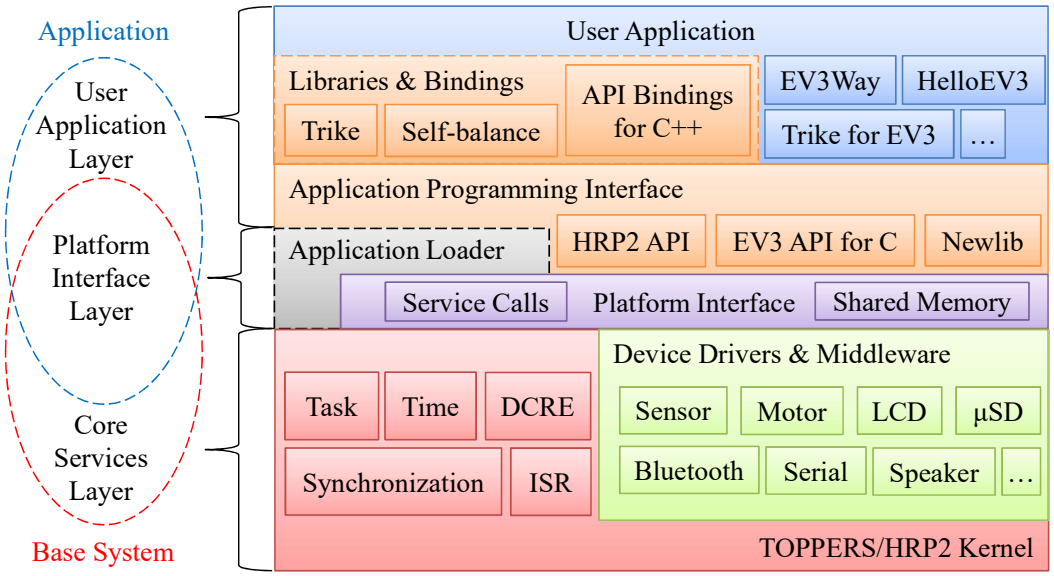

Fig. 2.2 Architecture of EV3RT

in standalone mode can be directly booted by EV3 and executed under both privileged mode and non-privileged mode. It is useful for building a platform or firmware as developer has full access to RTOS kernel and hardware resources in this mode. In fact, the base system of dynamic loading mode itself is an application in standalone mode. mruby on EV3RT + TECS [127], a platform for programming EV3 in mruby, is also developed in this mode.

EV3RT uses a layered architecture that decouples user application from the infrastructure software as shown in Fig. 2.2, in order to achieve high flexibility, reliability and extensibility. This architecture is mainly applied to dynamic loading mode but can also be considered as a reference model for standalone mode. It consists of three layers as follows:

Core Services Layer: This layer is to provide services needed by applications and monitor the running application. It mainly consists of TOPPERS/HRP2 kernel, device drivers and middleware. TOPPERS/HRP2 kernel is a static RTOS kernel with protection functionalities. EV3RT uses it to provide system services with high reliability. Potential defects and bugs in user application (e.g. illegal memory access) will not harm the platform. Instead, they are monitored and handled by this layer and will be logged to provide information for debugging. Device drivers and middleware are modules to support various features of EV3, such as PWM (pulse-width modulation) motor control and file system. They are mainly running under privileged mode since they usually wish to manipulate hardware devices directly. EV3RT console, a user interface for selecting application to launch or viewing system logs as shown in Fig. 2.3, will show at startup after all services have been initialized.

Fig. 2.3 Screenshot of EV3RT console

Platform Interface Layer: This layer acts as an interface between core services layer (CSL) and user application layer (UAL). It specifies a list of functions that must be implemented in CSL. These functions are wrapped as service calls so that they can be called from user application which runs under non-privileged mode. There are some service calls to obtain the pointer of a shared memory area. Shared memory can be used as a simple interface to eliminate the overhead of calling the same service call repeatedly. Examples of shared memory include read-only sensor data and read-writable LCD frame buffer. Data structures and macros shared between CSL and UAL are also defined in this layer. User applications are only dependent on this layer so they can be compiled and linked as loadable modules without any detail and code of CSL. Even if implementation of CSL changed a lot, modifications to source code of UAL are unnecessary as long as this layer does not change. Therefore, platform interface layer can be considered as the application binary interface (ABI) of EV3RT and is given a version number called PIL version. This layer also includes an application loader which can be used to update the running application dynamically. On launching an application, the loader checks the PIL version of the base system and the application at first to make sure they are compatible (with the same version). The running application can be manually terminated using EV3RT console, and if it is crashed, the loader will automatically unload it.

User Application Layer: This layer consists of software components for building user applications and the user application itself. APIs for C language are provided, which will be introduced later in the next subsection. Runtime and API bindings for C++ have been implemented to support object-oriented programming. In addition, libraries for common robot functionalities such as self-balancing are also available. Some sample applications are included to help developers get started with EV3RT. All code in this layer runs under non-privileged mode inside a determined protection domain called `TDOM_APP`, which will

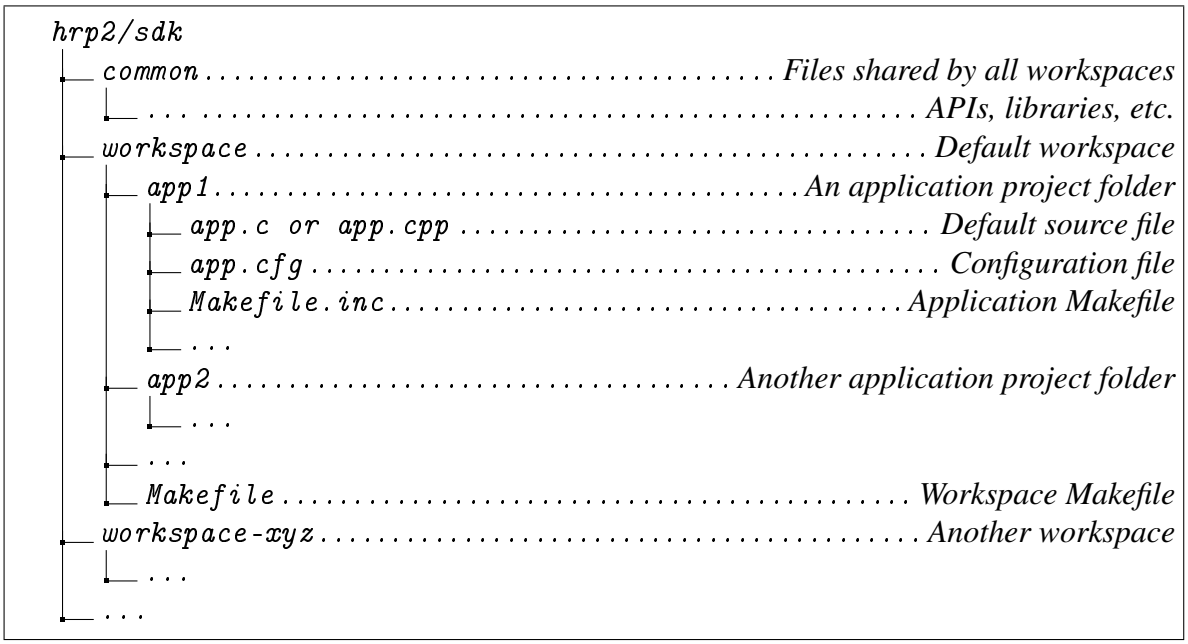

Fig. 2.4 Structure of the sdk folder

be explained later in Section 2.3. Currently, only one user application can be running at the same time on EV3RT.

2.2.2 Application Development in EV3RT

Users can develop applications for EV3RT under Linux, Mac OS X or Windows. EV3RT uses GNU Arm Embedded Toolchain [11] for compiling. All the required software packages can be easily installed by following the guide on our website [44, 43]. Installation of EV3RT to the intelligent brick can be done by just putting boot image of the base system, which can be built with one single command, into the root directory of EV3's microSD card. In this subsection, some important information about developing applications in EV3RT are described.

2.2.2.1 Application Project Management

In EV3RT, user application projects are managed inside the `sdk` (short for software development kit) folder. The `sdk` folder includes a `common` folder holding common files such as source files of APIs and libraries, a default workspace folder named `workspace` and optional workspace folders which can be created by users, as shown in Fig. 2.4. A workspace folder is a folder used to hold multiple application projects. Each application project corresponds to

```
APPL_COBJS += balancer.o \
    balancer_param.o

APPL_CXXOBJS +=

SRCLANG := c++

ifdef CONFIG_EV3RT_APPLICATION

# Include libraries
LDIR = $(EV3RT_SDK_LIB_DIR)
include $(LDIR)/libcpp-ev3/Makefile

endif
```

Fig. 2.5 Example of application-specific Makefile

a dedicated folder under some workspace folder. All the sample applications can be found in the default workspace.

With the workspace feature, application projects can be managed in an easy way. A new application project can be simply created by duplicating a existing project folder with a new name. Deleting a project is also straightforward: just remove the folder. An application can be built in two modes, as mentioned above, with its folder name. For example, for an application project folder named `app1`, developer can build it in dynamic loading mode by executing `make app=app1` command under its workspace folder. After building successfully, a loadable module named `app` will be generated. Further, developer can also use `make img=app1` to build it in standalone mode. In that case, a boot image file named `uImage` will be generated.

An application project has three files by default, `app.c` (or `app.cpp` when developing in C++), `app.cfg` and `Makefile.inc`. `app.c` (or `app.cpp`) is the default source file of an application which will be compiled automatically. `app.cfg` is the configuration file of an application. An application can be built in dynamic loading mode, only if all kernel objects defined in `app.cfg` belong to the predetermined application protection domain `TDOM_APP`. `Makefile.inc` is the application-specific Makefile. Application build configuration such as source files, libraries, programming language can be customized by modifying this file. See Fig. 2.5 for an example of `Makefile.inc`.

2.2.2.2 Development Process

The process to develop an application in standalone mode is listed as follows.

1. Power on the intelligent brick
2. Write code for the application
3. Generate the boot image `uImage`
4. Connect EV3 to PC with a USB cable
5. Copy `uImage` to the root of the microSD card
6. Restart the intelligent brick
7. The new application will get executed
8. Application needs modification, go to step 2

It should be noted that every time the application is modified, it is required to reboot the EV3 brick and write the microSD card with a PC. This procedure is a bit annoying and, if the application uses Bluetooth, the need of reconnection will make it even worse. Therefore, the standalone mode is not recommended except for developing a firmware or platform.

Developing an application in dynamic loading mode can be much more smooth. After the EV3 is booted, EV3RT console will show up on the display. Developers can use the loader in EV3RT console to receive and execute a new application wirelessly transferred from Bluetooth. Besides, applications can also be put into microSD card for execution via USB. If an application is running, user can press the back button for about 0.5 second to show the EV3RT console to terminate it and load a new application. The process is listed as follows.

1. Power on the intelligent brick
2. Write code for the application
3. Generate the loadable application module
4. Upload application via Bluetooth or USB
5. Start the new application
6. Application needs modification, go to step 2

2.2.2.3 APIs for User Application

EV3RT currently provides three types of APIs as follows:

TOPPERS/HRP2 Kernel API: This API allows user applications to access RTOS services provided by HRP2 kernel. It includes μITRON-like [132] static APIs and service calls, whose details are described in the TOPPERS new generation kernel specification[128]. Static APIs are used in the configuration file to configure the kernel objects of an application statically. On the other hand, service calls, which may be called system calls in other operating systems, are used in source files to request services at run time. In the HRP2 kernel specification, cyclic handlers can be only created in the kernel domain, which run in interrupt context under privileged mode without protection. Since cyclic handlers are also very useful in user applications, EV3RT extends the original kernel API with user domain cyclic handler functionality.

C/C++ Standard Library: Newlib[103], a C/C++ standard library implementation for embedded systems, has been ported to EV3RT. With the standard library, users can develop applications for EV3RT as easily as developing a normal C/C++ application. The support of dynamic memory management is required for most functions in Newlib. However, the default memory allocator in Newlib only supports one single heap, which is not compatible with the protection model of HRP2 kernel, and can not provide any real-time guarantee. We replaced it with the TLSF (Two-Level Segregated Fit) memory allocator [85], a memory allocator designed for real-time embedded systems. Since memory objects in HRP2 kernel are configured statically, a fixed-size memory pool is preallocated for Newlib. EV3RT also integrates communication services into the standard file operations. Special files for communication devices such as Bluetooth or serial port can be obtained. Developers can use these files to do communications just like accessing a file, by calling functions such as `fprintf()` and `fgetc()`.

EV3RT C Language API: This API provides C language functions to support hardware devices such as sensors and other platform-specific features. Since it is used to control the robot actually, most of its functions provide high real-time performance, except those involving microSD card operations. The supported devices and features are listed as follows:

- EV3 brick (buttons, LED, LCD and speaker)
- Memory file and microSD card
- Bluetooth and serial port
- Servo motors and rotary encoder

- Various sensors (ultrasonic, color, etc.)

These APIs allow users to develop applications easily without any knowledge about the implementation details of core services layer (e.g. device drivers) in EV3RT. If a developer wants to access device drivers (including middleware) directly, the standalone mode must be used, since device drivers run under the privileged mode. In that case, the developer will also need to learn the usage of device drivers (for example, from their documents).

Besides above APIs, optional static libraries like API bindings for C++ are also available.

2.2.2.4 Sample Applications

Several sample applications come with EV3RT to help developers get started. Two representative samples are introduced as follows:

`HelloEV3` is a program which can be used to test the features of EV3RT thoroughly. It provides a simple GUI which can be operated with buttons on the intelligent brick. This graphical menu contains tests for all functions provided by EV3RT C language API and itself is also implemented with those functions. Further, user is allowed to reconfigure the connection of devices such as sensors and motors dynamically, without recompilation or reboot. With this sample, the implementation of EV3RT can be checked easily and developers can learn how to use EV3RT APIs by referring to its source code.

`EV3Way` is a sample application to control a two-wheeled self-balancing robot. The researches on two-wheeled self-balancing robot (a.k.a two-wheeled inverted pendulum mobile robot) have gained momentum over the past decades and shown that high real-time performance is required [136]. This application is developed by the executive committee of ET Robocon in C++ to control a robot called EV3Way-ET. The reference construction of EV3Way-ET is available on GitHub [42]. There are two tasks, the balancing task and the communication task, in this application. The balancing task runs in a high priority to control the robot with the self-balancing algorithm and the communication task runs in a lower priority to communicate with user via Bluetooth. Both of the tasks can work stably, which shows that EV3RT is suitable for developing applications with high real-time requirements.

2.3 TOPPERS/HRP2 RTOS Kernel

TOPPERS/HRP2 kernel (or just HRP2 kernel) is used by EV3RT as its RTOS kernel. It is a μITRON-like [132] open source RTOS kernel created by the TOPPERS project [129]. HRP2 is short for High Reliable system Profile version 2, and by the way the first version of

HRP kernel has been adopted by the HII-B rockets [59] and proven its excellent real-time performance and reliability.

Since nxtOSEK/JSP [31], a popular RTOS platform for NXT, also uses RTOS kernels from the TOPPERS project, developers of NXT should find it easy to migrate to EV3 with EV3RT. Aside from the consideration of migration, the most important reason for choosing HRP2 kernel is that it is a static RTOS supporting various protection functionalities. Mindstorms EV3 is a relatively large-scale embedded system, and if its software platform is not protected, a bug in user application may easily break the whole platform, which makes debugging very difficult.

In this section, we focus on explaining how features of HRP2 kernel are applied to make EV3RT a reliable software platform. For a thorough description of HRP2 kernel, check the TOPPERS new generation kernel specification [128].

2.3.1 Static Configuration Approach

A static operating system kernel is a kernel whose kernel objects (or resources), such as tasks and memory areas, are configured at design time and statically allocated at compile time. Both EV3RT and the user applications on it are developed following this static configuration approach.

In HRP2 kernel, developers statically configure the kernel by writing configuration files using static APIs. See Fig. 2.6 for an example of configuration file. A tool called configurator will parse these configuration files. By using the parsing results as input, the configurator interprets some template files to generate necessary C source files or linker scripts. Template files are written in a template language defined by HRP2 kernel for generating files. Common template files such as generating source files for kernel objects are included in HRP2 kernel. Target-dependent template files like generating linker scripts or translation tables to support memory protection are also available for many popular targets. However, the processor of EV3 was not supported by HRP2 kernel, and thus we implemented those target-dependent template files for it.

Research has shown that a static OS design is more reliable and can provide a significantly better resilience to soft errors than the dynamic ones [58], as many potential software defects can be found during the configuration process. Besides, a static OS also tends to have higher and more stable performance and consume less memory.

```
KERNEL_DOMAIN {
  /* create a system task */
  CRE_TSK(MAIN_TASK,{ TA_ACT, 0, maintsk, 6, 1024, NULL });
  /* register a memory object */
  ATT_MOD("sample_kernel.o");
  /* define an extended service call */
  DEF_SVC(SVC1, { TA_NULL, svc1_entry, 64 });
}
/* a user domain named DOM1 */
DOMAIN(DOM1) {
  /* create a user task */
  CRE_TSK(TASK1, { TA_ACT, 0, task1, 6, 1024, NULL });
  /* register a memory object */
  ATT_MOD("sample1.o");
}
/* a user domain named DOM2 */
DOMAIN(DOM2) {
  /* create a user task */
  CRE_TSK(TASK2, { TA_NULL, 0, task2, 6, 1024, NULL });
  /* create a semaphore named SEM1 */
  CRE_SEM(SEM1, { TA_NULL, 0, 1 });
  /* configure access rights of SEM1 */
  SAC_SEM(SEM1,{ TACP(DOM2), TACP(DOM1)|TACP(DOM2), \
    TACP(DOM2), TACP(DOM2) };
}
/* define a shared memory object and set access rights */
ATA_SEC(".appheap", { TA_NULL, "RAM" }, \
  { TACP_SHARED, TACP_SHARED, TACP_SHARED, TACP_SHARED });
```

Fig. 2.6 Example of a configuration file

2.3.2 Kernel Object Access Control

A kernel object is a system resource managed by the RTOS kernel. Kernel objects are defined in and will be generated from configuration files. Each kernel object created will be associated with an ID which can be used to access it. Tasks, semaphores and data queues are typical examples of kernel objects.

The concept of protection domain is introduced to support access control of kernel objects. There are two types of protection domains in HRP2 kernel: kernel domain and user domain. The kernel domain is unique while there can be multiple user domains. A task must belong to some protection domain to determine its permissions. Tasks in the kernel domain, called

system tasks, are executed under privileged mode which has full access rights to all kernel objects. Tasks in a user domain, called user tasks, are executed under non-privileged mode with limited access rights. Other runnable kernel objects, such as alarm handlers, are always executed under privileged mode and thus must belong to the kernel domain. A protection domain is the finest granularity to grant access rights of a kernel object, which means that tasks in the same protection domain have the same access rights. By default, a kernel object allows user tasks in the same protection to access it. A non-runnable kernel object like semaphore can also be shared by all protection domains without belonging to one of them, which allows any task to access it.

In order to configure access permissions more flexibly, operations to each type of kernel objects are divided into four groups: type 1 operations, type 2 operations, management operations and reference operations. For example, in the case of semaphores, type 1 operations are for signaling, type 2 operations are for waiting, management operations are for configuring access rights, and reference operations are for acquiring status. The access rights of each group of operations for a kernel object can be configured individually.

In EV3RT, the core services layer mainly works in the kernel domain while all kernel objects in the user application layer belong to a determined user domain called `TDOM_APP`. The core services layer does not allow user application to access its objects directly. Instead, service calls and shared memory defined in the platform interface layer must be used. In this way, kernel objects in the base system are protected from illegal or undesired access caused by the user application.

2.3.3 Memory Protection Support

Memory protection is necessary because user application can still harm the base system by operations such as writing improper values to a hardware register or changing data in the kernel, even if kernel objects are protected. What is worse, if user application breaks the file system, all data on the microSD card might be corrupted. As a high reliable RTOS, HRP2 kernel does support memory protection, with static configuration of memory objects.

In HRP2 kernel, a memory object represents an area of memory controlled by the kernel. Lots of information is associated with a memory object, including base address, size and attributes of that area, and access rights granted to protection domains. An attribute is a property that always holds regardless of which protection domain it is accessed from. For example, if a memory object has the attribute `TA_NOWRITE`, its memory area cannot be written even by a system task. There are also attributes like `TA_UNCACHE` to control cache behavior. If base address and size of a memory object is fixed, such as a hardware register, developer can use the `ATT_MEM` static API to create it explicitly. In the case of text and data section in

an object file whose base address and size are unknown at design time, the `ATT_MOD` static API can be used to generate memory objects from an object file with specified access rights. Configuring attributes and permissions for a named section defined in the source code is also supported, by using the `ATT_SEC` static API with the section name. Memory objects should never overlap, otherwise an error message will show up during configuration.

Although HRP2 kernel is designed to support memory protection with both memory protection unit (MPU) and memory management unit (MMU), the ARMv5 MMU used by EV3 was not supported. In our previous work [77], we have supported the ARMv5 MMU by designing an algorithm to generate ARMv5 page tables for each protection domain, and proposed a method to optimize the TLB flushing overhead. With that optimization, ultra-low latency of context switching can be achieved.

For memory objects in the base system of EV3RT, only those defined as shared memory in the platform interface layer can be accessed from the user application layer. Most of the shared memory areas are read-only, such as sensor values and font data. Only memory areas that will not break the base system even if they are corrupted, like LCD frame buffer for user application, can be configured as read-writable shared memory. The base system has its own frame buffer which is protected from user application, for displaying interface of core services like EV3RT console. In this way, shared memory can be used as an efficient and safe method of passing data in EV3RT.

2.3.4 Extended Service Call

Although shared memory is very efficient, it is not enough because most user applications will also wish to execute code under privileged mode, such as using PWM device to control a motor, in a safe way. HRP2 kernel provides the extended service call functionality to support this case.

The term service call in HRP2 kernel has a similar meaning to system call in Linux. Service calls act as an interface between user domains and the kernel. This interface is essential to provide system services for a user task which is usually prevented from directly manipulating the kernel's memory. When a service call is called, it is executed under privileged mode and aware of the caller's protection domain, so illegal operations can be blocked by checking the access rights.

As a general-purpose operating system, Linux has a stable system call interface, and device drivers are abstracted as special files to be accessed with file I/O system calls. Access permissions of device drivers in Linux are also controlled using file system. Although this approach brings excellent portability, it also introduces overhead of the file system which influences the real-time performance. In HRP2 kernel, instead of providing a universal

```
KERNEL_DOMAIN {
  DEF_SVC(TFN_MOTOR_COMMAND,{ TA_NULL, \
    extsvc_motor_command, 1024 });
}
```

a. define extended service call in configuration file

```
ER_UINT extsvc_motor_command(intptr_t cmd, intptr_t size,
  intptr_t par3, intptr_t par4, intptr_t par5, ID cdmid) {
  ...
}
```

b. implement service call in kernel domain

```
ercd = cal_svc(TFN_MOTOR_COMMAND, cmd, size, 0, 0, 0);
```

c. call service call from user domain

Fig. 2.7 Example of extended service call

interface for device drivers, the functionality of conveniently defining new service calls, called extended service calls, is supported.

Fig. 2.7 is an example to show how extended service calls are used in device drivers and middleware of EV3RT to provide services safely. Firstly, a new service call is defined with the static API `DEF_SVC`. A unique number (`TFN_MOTOR_COMMAND` in the example), called function code, is associated with each service call. The function name and stack size for the service call must also be specified. Thereafter, function body of the service call, which will be executed under privileged mode, is implemented. The function has 6 parameters: 5 of them are passed from the caller and the last one (`cdmid`) holds the protection domain ID of the caller. These parameters are carefully checked and an error code will be returned if any problem occurs. In user application, the `cal_svc()` API is used to call a service call with its function code and arguments. The context will enter privileged mode to execute corresponding function and switch back with return value after the function has completed.

2.3.5 Task Priority Protection

With above policies applied, the space of base system can be protected from defects in user application. However, a user application can still influence the base system in the aspect of CPU time. HRP2 kernel uses priority-based preemptive scheduling algorithm. If a user application changes the priority of a task, for example, of an infinite loop to the highest priority, the intelligent brick will be stuck forever until a physical power-off is performed, because all tasks in the base system cannot be scheduled.

In order to solve this issue, we added the `LMT_DOM` static API to the HRP2 kernel specification. This API can limit the highest priority that can be set at run time for tasks in a protection domain. It does not limit the initial priority of tasks since they can be checked during configuration, which allows developers to control the system more flexibly. EV3RT uses this API to limit task priority in `TDOM_APP`, and thus the base system will not be preempted by tasks in user application.

Besides, Bluetooth service in the base system has another kind of issue on task priority. The CPU utilization of task to process Bluetooth packets depends on the speed of traffic. If the task has higher priority than user application and packets come in a very high rate, it will have a negative effect on the real-time performance of user application. On the other hand, if the task has lower priority than user application, it may never get scheduled, which will break the Bluetooth service. EV3RT copes with this issue by using a QoS (quality of service) task to control the priority of Bluetooth task dynamically. The QoS task will periodically raise the priority of Bluetooth task, which usually works in the lowest priority, to a priority higher than user application for a short time.

By default, the Bluetooth task will run in high priority for 1 ms in every 20 ms. These timing parameters are decided by an experimental approach to find the minimum required CPU resource for a relatively stable Bluetooth connection. During the experiment, a task of infinite busy loop is running as user application. The exection time of Bluetooth task in high priority is fixed at a single system tick, which is 1 ms. We then change the period to control CPU resource given to the Bluetooth task, and check whether the Bluetooth connection is stable. A connection is considered to be stable if it can complete pairing process and stay connected for over one hour. After having tested and failed with 100 ms ($\approx$1% CPU), 50 ms, 30 ms, 25 ms periods, we found that 20 ms period can provide a stable connection. Meanwhile, as a representative application with high real-time requirements, the balancing task of `EV3Way` has an execution time of 1 ms with 5 ms period. Therefore, Bluetooth service can be kept alive with this policy while has very limited influence on real-time performance. Users are also allowed to change the timing parameters of QoS policy or disable it as necessary.

2.4 Dynamic Module Loading

Dynamic updating mechanisms of software system or applications without reboot are becoming significantly necessary to reduce the time to market and development costs, with the rapid growth of the complexity of embedded systems. System reboots can be slow and disruptive and will drop all current status like network connections, which are very expensive for some

safety-critical applications such as avionics [116]. For Linux-based software platforms, new applications can be transferred and executed on-the-fly within a few seconds via network protocols such as Secure Shell (SSH) [142]. While the static design of HRP2 kernel provides high robustness, the lack of dynamic updating support can be one of its main drawbacks.

In this section, we present a dynamic module loading mechanism for HRP2 kernel. The application loader in EV3RT is implemented by referring to this mechanism. Main characteristics of our original mechanism are listed as follows:

Excellent portability. The design of our mechanism, per se, does not depend on any specific hardware, although the motivation is to support dynamic application loading in EV3RT. It is able to support most of targets where HRP2 kernel works. Further, developers can apply it to existing systems very easily, without implementing any complex function such as dynamic memory or page table management.

Keep it simple, static. Our mechanism is to support dynamic updating of modules rather than to support development in a dynamic way. HRP2 kernel benefits a lot from the simplicity of its static design, especially in the aspects of performance and robustness, and our design is able to keep those advantages. All loadable modules are still configured and developed in the same way of developing a normal HRP2 application, which is static, with some restrictions and minor changes. Thus, resources required by a module are statically determined at compile time and can be checked at load time.

Reliable and fault-tolerant. Modules may contain bugs and could lead to incorrect or unexpected behaviors after loaded. Our mechanism has the feature to guarantee that the modules loaded will not harm the whole system. The loader will verify the configuration information of a module before loading, and the unloading of a module can be performed in an elegant way too. Further, protection functionalities introduced in previous section can also be applied to save the system from run-time failures of modules.

High real-time performance. High real-time performance is required as a dynamic module loading mechanism for hard real-time operating systems like HRP2 kernel. Since loadable modules in our mechanism are developed in a static way and generated as native code, the real-time performance can be easily guaranteed after loaded. Moreover, loading and unloading operations are provided as service calls working under task context and will only lock the CPU for a very few cycles. That is to say, using our mechanism has very little influence on the real-time performance of the whole system.

2.4.1 Dynamic Creation Extension

Although HRP2 kernel requires all kernel objects and resources to be allocated statically according its static design, it does provide a kernel extension, called Dynamic CReation

Extension (DCRE), that allows kernel objects to be dynamically created or deleted at run time.

DCRE still follows the static methodology of HRP2 kernel by using the object pool design pattern [113]. The maximum number of each kind of kernel object that can be created must be predefined in configuration files at design time. Objects pools for these kernel objects will be generated at compile time and the resources (e.g. data structures like task control blocks) required by them will be allocated statically. During the startup process of kernel, all kernel objects in pools will be set to the 'not in use' state (or the invalid state). An object can be created by obtaining a 'not in use' object from the pool and initializing it to 'in use' state (or the valid state). When an object created dynamically is no longer needed, it can be deleted and returned to its pool by setting back to the 'not in use' state with necessary clean-up process performed. In this way, kernel objects can be created and deleted without dynamic allocation of memory or other resources.

DCRE is designed in the spirit of minimality principle that it does not depend on any prediction of hardware or application. Consequently, the mechanisms for dynamic memory management are not defined by DCRE. However, dynamic memory management is necessary for dynamically creating tasks whose stacks have different size, address and access permissions which can not be determined and allocated statically. Thus, dealing with dynamic memory management is required for designing dynamic module loading mechanisms based on DCRE.

2.4.2 Design of Proposed Mechanism

In this subsection, design of the reference model of our dynamic module loading mechanism is described. Our mechanism mainly consists of three kinds of concepts: loadable modules, containers and the loader. Fig. 2.8 provides an overview of these concepts.

Loadable modules are developed by module developers and can be built as binary files containing information for loading and execution. Containers, whose protection policy, memory layout and resource limit can be configured, are kernel objects used to hold a loadable module. The loader is a system service that is responsible for checking and placing a loadable module into a container, preparing it for execution and unloading it when it is no longer needed. System developers can create and configure containers in the base system and use the loader to load module into specific container dynamically. The reference model of these concepts is explained in following subsections.

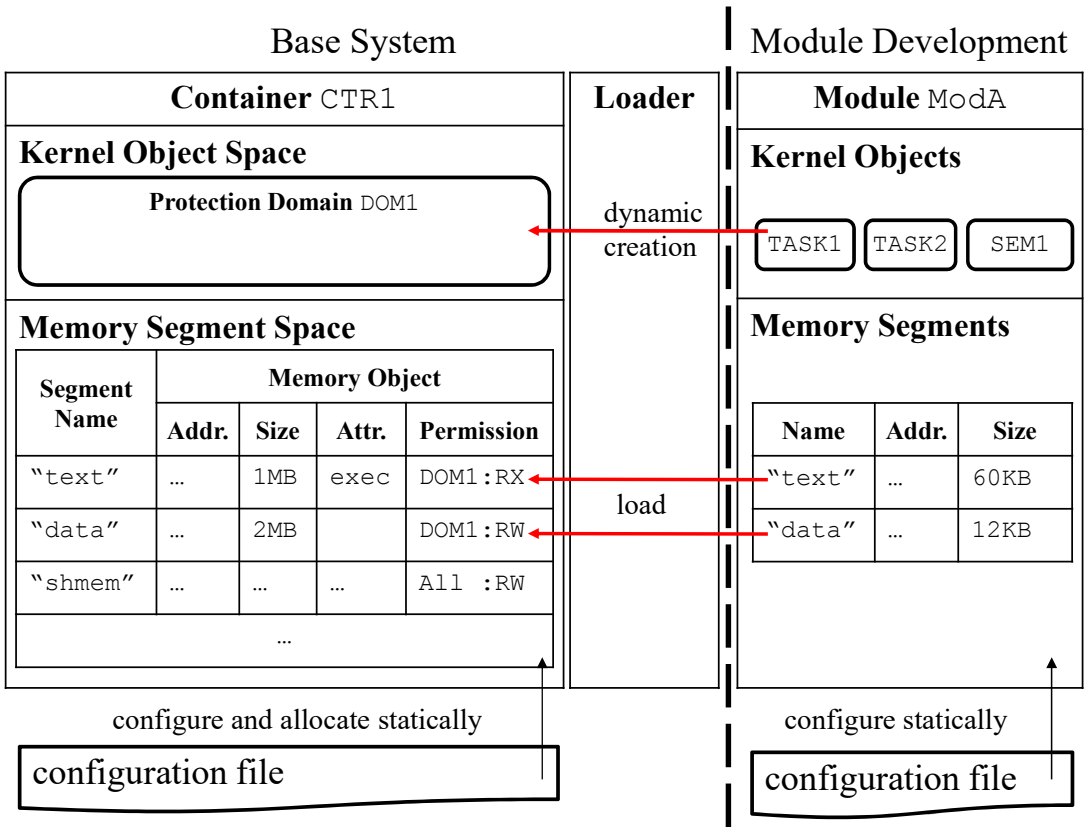

Fig. 2.8 Proposed Dynamic Loading Mechanism

2.4.2.1 Loadable Module

A loadable module (or just module) is a software component that can be dynamically installed into the existing system to provide extended features such as new services, applications or even device drivers. A module installed can be unloaded in order to free the resources allocated to it such as memory, when the functionality provided by it is no longer required.

Logically, a module is a collection of kernel objects and data. All the kernel objects must be defined in a configuration file of the module at design time. Module developers are not allowed to perform any dynamic creation or deletion of kernel objects during the execution of a module. In this way, the resources required by a module can be statically determined at compile time and will be provided by the loader at load time.

Unlike similar concepts in general-purpose operating systems such as loadable kernel modules (LKM) in Linux [109] or kernel-mode drivers in Windows NT [14], which are running in privileged mode without protection, our modules are supported to work in both privileged mode and non-privileged mode with protection.

In order to achieve high reliability while keeping the simplicity, there are some constraints on the development of modules:

Kernel objects must belong to the same protection domain. Access permissions of kernel objects are granted in the granularity of protection domain. A module will be loaded into a container, which is associated with a single protection domain, in the base system for execution. Permissions granted to a module depend on the protection domain of its container. Therefore, all kernel objects in a module must be configured to belong to the same protection domain at design time. The protection domain of a module is usually omitted, in which case the protection domain of container will be used at loading. If the protection domain is specified explicitly, the base system is responsible for choosing a suitable container. It should be noted that the protection domain of a module must not be decided freely by the module developers themselves, which can make any protection policy become meaningless. If the base system is intended to support explicitly specified protection domain of a module, verification schemes such as storing digital signatures in the module for checking trusted developers should be introduced at the implementation level of our mechanism. The loader in EV3RT does not support signature verifying since its user applications will only be loaded into the predetermined protection domain (i.e. `TDOM_APP`).

Configurations of memory objects are not allowed. Implementing dynamic memory management with protection is not required to support our mechanism for the simplicity of porting. All the memory protection information such as page tables will be statically generated at compile time of the base system. Since configuring memory objects may result in changing of the memory protection information, it is not allowed in the development of modules. Instead, we provide an approach, which will be described later in the section of container, to controlling memory protection for a module by putting memory contents into segments with access permissions specified by a container.

Dynamic creation extension (DCRE) cannot be used. The main purpose of DCRE is to support developing middlewares such as a dynamic loader, rather than to support developing user applications in a dynamic way. Therefore, DCRE does not provide fine-grained access control on managing objects. That is to say, if a protection domain has the permission to use DCRE, tasks in that domain can create objects belonging to any protection domain including the system domain, which may be used to bypass existing protection policy. Further, the allocation of kernel object pools will introduce configurations of memory objects implicitly, which will violate the second constraint. Hence, while the implementation of our mechanism is based on DCRE, using dynamic creation extension is not supported in the development of modules.

Physically, a module is an object file that contains the module information and multiple segments with code or data. The module information consists of a module configuration table and other implementation-specific or application-specific information. An example of the module configuration table is shown Fig. 2.9. Each entry in the module configuration table corresponds to a configuration entry in the configuration file, which represents a static API call such as `CRE_TSK`. An entry in the module configuration table has three members: a static API code, a pointer to the argument of that call and a pointer to store the return value. The loader will create kernel objects according to the static API code and the argument. The return value is used to pass information to the module. For example, in Fig. 2.9, the actual ID of TASK1 is unknown at compile time and will be set by the loader through the return value pointer after the kernel object of TASK1 is created.

The memory contents inside a module are divided into segments. There are at least two segments required to support our mechanism, the text segment and the data segment. The text segment is used to store instructions and read-only data, and the module is not permitted to write to the memory space specified by it. The module information must also be placed into the text segment for safety concerns. Static variables such as global variables and static local variables are contained in the data segment which is writable by the module. Stack areas of tasks are generated as static variables as shown in Fig. 2.9 and hence will be put into the data segment. A module may include other segments, specified by its container, like shared memory segment for finer memory management. Most compilers support to place specific memory contents into a particular segment. For example, the GNU Compiler Collection (GCC) [53] allows user to set the section name for variables in source code and put multiple sections into specified segment in linker script.

```
// Content in the configuration file of a module:
// CRE_TSK(TASK1,{TA_ACT,0,task1,TMIN_TPRI,STACK_SIZE,NULL});

ID _module_id_TASK1 __attribute__((section(".module.text")));
static STK_T _module_ustack_TASK1[COUNT_STK_T(STACK_SIZE)];

/* Structure of information used to create a task */
typedef struct t_ctsk {
    ...
} T_CTSK;

/* Array of configuration information for tasks */
static const T_CTSK _module_ctsk_tab[1] = {
    { TA_ACT, 0, task1, TMIN_TPRI, ROUND_STK_T(STACK_SIZE),
        _module_ustack_TASK1, DEFAULT_SSTKSZ, NULL },
};

/* Structure of a module configuration entry */
typedef struct {
    FN          sfncd; /* Static API code */
    const void *argument;
    void       *retvalptr;
} MOD_CFG_ENTRY;

/* Array of a module configuration entry */
const MOD_CFG_ENTRY _module_configuration_entries[] = {
        { TSFN_CRE_TSK, &_module_ctsk_tab[0],
        &_module_id_TASK1 },
};

/* Structure of the module configuration table */
typedef struct {
    const MOD_CFG_ENTRY *entries;
    const SIZE           entry_number;
} MOD_CFG_TAB;

/* The module configuration table */
const MOD_CFG_TAB _module_configuration_table =
    { _module_configuration_entries, 1 };
```

Fig. 2.9 Example of a module configuration table

```
DML_CRE_CTR(CTR1, DOM1, "text", "data");
DML_ATA_SEG(CTR1, "text", { TA_EXEC, NULL, 1 * 1024 * 1024, NULL }, \
    { TACP(DOM1), TACP_KERNEL, TACP_KERNEL, TACP_KERNEL });
DML_ATA_SEG(CTR1, "data", { TA_NULL, NULL, 2 * 1024 * 1024, NULL }, \
    { TACP(DOM1), TACP(DOM1), TACP_KERNEL, TACP_KERNEL });
```

Fig. 2.10 Example of configuring a container and its segments

2.4.3 Container

Dynamic resource allocation is not supported in HRP2 specifications: all resources must be configured at design time and allocated at compile time for generating the base system. This means that the available resources of a module must also be defined statically at the design time of the base system. A new concept called container is introduced to support this.

A container is a kernel object acting as the placeholder of a module. Configuring a container allows system developers to control the characteristics of modules loaded into it such as their protection policy, memory layout and resource limit (e.g. the maximum number of each kind of kernel objects that can be created). We define three kinds of static APIs to configure a container:

DML_CRE_CTR(ctrid, domid, text_segment_name, data_segment_name): A container can be created with a specific container ID (`ctrid`) using this API. It is associated with a protection domain (`domid`). All kernel objects created by loading a module into the container will belong to that domain. In such a way, system developers can control the access permissions of a module by configuring the protection domain of its container. This API also specifies the name of text segment (`text_segment_name`) and data segment (`data_segment_name`) in a module. These two segments are mandatory for a module.

DML_ATA_SEG(ctrid, segment_name, { mematr, base, size, paddr }, { acptn1, acptn2, acptn3, acptn4 }): A memory segment can be attached to a container (`ctrid`) using this API. Each segment in the same container must have a unique name (`segment_name`) which can be used in the development of modules. A memory object with specific attributes (`mematr`), address (`base` and `paddr`), size (`size`) and access permissions (`acptn1-acptn4`) will be created for a segment. In this way, the memory protection information of each segment can be configured at design time of the base system. It means that the memory area for a segment is preallocated at compile time and has a maximum size. When loading a module into a container, segments in that module will be loaded to corresponding memory area with the same segment name. See Fig. 2.10 for an example of configuring segments with this API. In this example, the text segment is 1MB size, executable (`TA_EXEC`) and read-only to `DOM1` which is the protection domain of its container. The data segment is 2MB

size and both readable and writable to `DOM1`. It should be noted that the addresses (both virtual and physical) of these segments are omitted in the example. In this case, the base addresses of text and data segment will not be fixed and loadable module must be compiled to position-independent code that data segment has an unfixed offset with text segment [111]. By this API, the memory layout with protection information of modules supported by a container can be configured flexibly and the required resources for a container to load a module will be allocated statically.

DML_MAX_yyy(ctrid, max_number): For each kind of kernel objects, their maximum number inside a container can be limited by this series of static APIs. For example, `DML_MAX_TSK(CTR1, 10)` means that the container `CTR1` can only load a module with no more than 10 tasks. Using this API is optional, and if a kind of kernel object is not limited by this API, its maximum number will depend on the status of the kernel object pool.

2.4.4 Loader

The loader provides services to load a loadable module into a container and unload a module when it is no longer needed. The process of loading a module is listed as follows:

1. Verify the loadable module
2. Copy segments into the container
3. Perform relocation when necessary
4. Do hardware-dependent post-processing
5. Create kernel objects and resolve their IDs

The resources required by the module to be loaded can be known from its configuration table. The loader must verify it before doing any actual work for loading. If the module has a memory segment whose name does not exist in the segments of the target container, it cannot be loaded into that container. The size of every segment in the module must not exceed the maximum size of the corresponding segment in the container too. If the container has limits on the number of kernel objects that can be created, it must also be checked. It must be noted that the creation of kernel objects may still fail, depending on the status of the base system (e.g. available resources in kernel object pools). The loader should also make sure the module information is located in the text segment since it will be used for unloading and must not be changed by the module.

After verification, the loader copies each memory segment of the module into the corresponding segment in the container. Since memory for segments of containers is allocated statically, no dynamic memory allocation is needed in this step.

The base address of segment in a container may be not fixed. The loader must relocate segment in the module if its base address does not match the container's. In that case, the module must also include the relocation table created by the linker [74].

Modifications of text segment, including copying and relocating, involve indirect self-modification code, and may require post-processing on some hardware. For example, on ARM architecture [8], data and instruction cache are isolated. The cache synchronization such as flushing data cache and invalidating instruction cache for the modified text segment must be explicitly performed by the loader before further processing.

After handling memory segments, the loader can finally create kernel objects according to the configuration table of the module. For each kernel object created, the loader will pass its actual ID to the module through the return value pointer in the module configuration entry. The implementation of loader must guarantee atomicity for this step: if a kernel object is not created successfully, all kernel objects created previously must be deleted before any code in the module is executed.

IDs of all the kernel objects created for a module are stored in its configuration table, which is read-only to the module, after loaded. If the module is no longer needed, the loader can safely unload it by using information in that table to delete all its kernel objects.

2.5 Application Loader in EV3RT

Application loader in EV3RT is developed as a prototype implementation of the proposed dynamic module loading mechanism. It can dynamically load application from microSD card or Bluetooth without rebooting EV3. User can also put new applications into microSD card for loading by connecting EV3 to PC with a USB cable.

Fig. 2.11 shows the configuration of the application container in EV3RT. In EV3RT, the application container `APP_CTR` is associated with the application domain `TDOM_APP`. It has only two segments, the text segment and the data segment. Both of these segments do not have a fixed base address so relocation is required. The text segment is set to be read-only for all user domains (`TACP_SHARED`), not just the application domain. The data segment is shared between all user domains too. This configuration is based on the fact that EV3RT has only one user domain, the application domain. Therefore, all memory objects in user domains can be considered as global. This trick can reduce the number of non-global entries

```
DML_CRE_CTR(APP_CTR, TDOM_APP, "text", "data");
DML_ATA_SEG(APP_CTR, "text", \
    { TA_EXEC, NULL, TMAX_APP_TEXT_SIZE /* 1MB */, NULL }, \
    { TACP_SHARED, TACP_KERNEL, TACP_KERNEL, TACP_KERNEL });
DML_ATA_SEG(APP_CTR, "data", \
    { TA_NULL, NULL, TMAX_APP_DATA_SIZE /* 2MB */, NULL }, \
    { TACP_SHARED, TACP_SHARED, TACP_KERNEL, TACP_KERNEL });
DML_MAX_TSK(APP_CTR, TMAX_APP_TSK_NUM /* 32 */);
DML_MAX_SEM(APP_CTR, TMAX_APP_SEM_NUM /* 16 */);
DML_MAX_FLG(APP_CTR, TMAX_APP_FLG_NUM /* 16 */);
DML_MAX_DTQ(APP_CTR, TMAX_APP_DTQ_NUM /* 16 */);
DML_MAX_PDQ(APP_CTR, TMAX_APP_PDQ_NUM /* 16 */);
DML_MAX_MTX(APP_CTR, TMAX_APP_MTX_NUM /* 16 */);
```

Fig. 2.11 Configuration of the application container in EV3RT

in address translation tables, and thus optimize the overhead of TLB flushing[77]. Maximum number for each kind of kernel objects is also defined.

Applications in EV3RT are loadable modules. They start from a task instead of main function. In fact, an initialization task will be created for each user application. It is responsible for executing functions such as constructors of global C++ objects before starting other tasks in the user application. Since the base addresses of its segments are not fixed, applications will be compiled to generate complete position-independent code (with `-mno-pic-data-is-text-relative` option in GCC compiler) into Executable and Linkable Format (ELF) [87] files. Applications can only interact with the platform by service calls and shared memory areas. They do not directly depend on any address of the base system since the pointers of shared memory are acquired from service calls. In this way, recompilation of applications is not needed even if configuration of the application container (e.g. segment maximum size) is changed.

The loader is implemented to partially support ELF files for the ARM architecture [10], and configuration of the application container is hard-coded instead of parsed from configuration file. That is why we call it a 'prototype'. Although it can only properly relocate applications compiled with complete position-independent code, it is sufficient for supporting dynamic application loading in EV3RT. At present, the unloading of a module is done by killing all tasks in it forcedly and then deleting all kernel objects created for it.

Table 2.2 Software metric comparison of Mindstorms NXT and Mindstorms EV3

Lines of Code			nxtOSEK	lms2012
	Mindstorms	LCD	157	489
		Speaker	389	597
		Motor	137	3,176
		Sensor	235	5,620
		Soft I2C	421	1,598
		Soft UART	n/a	16,889
	SoC	SPI	135	919
		AVR	257	n/a
		SD	n/a	983
	Generic	FAT	n/a	5,375
		USB	664	16,746∼
		Bluetooth	330	14,186∼
		WiFi	n/a	34,763∼

2.6 Device Drivers and Middleware

Both hardware and software of Mindstorms EV3 have been improved vastly compared to its predecessor Mindstorms NXT. As a result, the scale of device drivers and middleware became one of the major challenges in developing EV3RT.

In this section, the strategy for implementing necessary device drivers and middleware for EV3RT is disscused firstly. Thereafter, the approach to reuse Linux drivers and how some middleware are integrated into EV3RT are explained.

2.6.1 Analysis and Strategy Decisions

To discuss the issue of scale in a quantitative way, we compared software metric of nxtOSEK [31] and lms2012 [72]. nxtOSEK is a popular platform for NXT while lms2012 is the code name of the stock Linux-based firmware of EV3. A utility called CLOC (Count Lines of Code) [2] is used to measure the amount of code. The statistics include both headers and source files with all comments and blank lines omitted. We have analyzed most of the important modules in these platforms and the results are listed in Table 2.2.

Software modules in Mindstorms can be mainly classified into three groups:

Mindstorms exclusive: This group includes drivers for those special-purpose devices such as motors and sensors. EV3 also uses its own software-based implementation for I2C and UART communication because its processor cannot provide adequate hardware resources. Although source code is available, detailed information for those devices is very limited.

SoC peripherals: This group includes drivers for those peripherals included in the SoC such as SPI and MMC/SD controllers. Specification and usage of those devices can usually be found in the data sheet and user manual released by the SoC maker.

Generic middleware: This group includes middleware of those common functions which can be found in various systems, such as file system and network. Functions supported by them are relatively advanced and it can be a heavy burden for users to control the low-level device directly. Thus, those middleware components are usually provided as a protocol stack or framework with a hardware abstraction layer to hide differences of hardware.

Mindstorms exclusive drivers of EV3 are about 20 times larger than NXT in the terms of total lines of code. Implementing these device drivers from scratch is extremely difficult and, if even possible, will take too much time. To reduce the workload of developing, we must reuse the existing source code as possible. Since lms2012 is a Linux-based firmware, we proposed an approach to reuse kernel-space Linux device drivers on HRP2 kernel.

For the SoC peripherals, we decided not to reuse the drivers from lms2012. Instead, the AM1808 StarterWare [122] released by Texas Instruments, the maker of EV3's SoC, is used. StarterWare is a free software development package including libraries and example applications for peripherals on the TI processors. It is designed for no-OS platform and can be easily integrated with EV3RT.

As to those generic middleware in lms2012, all of them are especially developed for a Linux system, which means that a near-complete compatibility layer will be required to reuse them directly. However, there are some open source alternatives for bare metal environment, since the functionalities they provide are actually very common even in simple embedded systems without a kernel. We ported BTstack [21] and FatFS [29] to HRP2 kernel so EV3RT can support Bluetooth communication and FAT file system.

2.6.2 Reusing Linux Device Drivers

Linux device drivers can be divided into two types by the execution modes: the user-space device drivers and the kernel-space device drivers. We analyzed their characteristics at first in order to reuse them in EV3RT.

A user-space device driver is an application or library whose code is, just as its name implies, running in user space. It is developed with the system call interface of the Linux kernel [82], which consists of about 380 system calls. Many user-space device drivers tend to use complicated system calls such as socket() and mmap(). Supporting those system calls on an RTOS kernel like HRP2 can be very difficult. User-space device drivers in lms2012 include Bluetooth protocol stack (BlueZ) and communication libraries between its virtual

machine runtime and low-level drivers. EV3RT does not reuse any user-space device driver from lms2012.

A kernel-space device driver is, on the other hand, a module that will be loaded into and then become part of the kernel. Since its code is running in kernel space, the user-space system call interface can not be used. Instead, it is developed with the Linux in-kernel API (a.k.a the Linux device driver interface) [81]. Although the Linux in-kernel API is not a stable interface, it is relatively simple compared to the system call interface. Except for those interfaces for generic middleware like USB and file system, the Linux in-kernel API just provides management functions for basic hardware resources such as timers and interrupts. Mindstorms exclusive drivers are mainly developed with those management functions as the devices they support are for special purposes. HRP2 kernel, like most RTOS kernels, also provides functions to manage the basic hardware resources and the similarities give us a chance to reuse Mindstorms exclusive drivers in EV3RT.

We have implemented a simplified compatibility layer of Linux in-kernel API, which enables the core parts of Mindstorms exclusive drivers to work on HRP2 kernel. Header files are needed in order to create a compatibility layer. We noticed that headers in Linux kernel will not interfere with headers in HRP2 kernel, and can be compiled easily. Thus, we just used those header files from Linux kernel. Functions supported in our compatibility layer, whose implementation details have been described in our previous work [77], are list as follows:

- Memory management
- IRQ (Interrupt ReQuest) handling
- Kernel locking: spinlock and semaphore
- High-resolution timer

As mentioned in section 2.3.4, device drivers in Linux use file system as their interface, which has relatively high overhead. Moreover, Mindstorms exclusive drivers are developed as loadable kernel modules but HRP2 kernel does not support dynamic loading of device drivers. Therefore, a device driver must be adapted for reusing in EV3RT. We describe the methodology to reuse a Linux driver with an example shown in Fig. 2.12. At first, the header file `driver_common.h`, which contains the compatibility layer we implemented, is included. Then, hacks needed to make the driver compiled or working properly are defined. For example, there are multiple modules in Mindstorms exclusive drivers having a symbol named `InitGpio`, we renamed them with a macro to avoid name conflicts. The original source file of the driver to be used is included after hacks. In principle, we do not modify

```
// Include the compatibility layer
#include "driver_common.h"

// Hacks for this module
#define InitGpio PWM_InitGpio
static void SetGpioRisingIrq(...) {
    ...
}
...

// Include source file to be reused
#include "d_pwm.c"

// Interfaces
ER_UINT extsvc_motor_command(...) {
    ...
    Device1Write(...);
    ...
}
...
```

Fig. 2.12 Example of reusing Linux device driver

the original source file for maintainability, but unneeded code like data structures used in dynamic loading is commented out. The interfaces of this driver are implemented at last. Basically, the file operation functions are wrapped to implement extended service calls defined in the platform interface layer. In such a way, a Linux device driver can be integrated into EV3RT and provide high real-time performance with overhead of file system eliminated.

Mindstorms exclusive drivers that have been successfully reused are listed as follows:

- PWM control for servomotors
- A/D converter for analog sensors
- Soft-based I2C ports for I2C sensors
- Soft-based UART ports for UART sensors
- UART sensor communication protocol
- LCD driver for graphics
- Speaker driver for sound

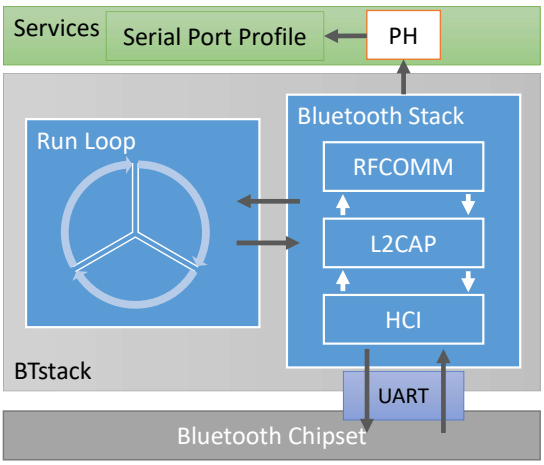

Fig. 2.13 Architecture of BTstack

We used CLOC [2] to evaluate the effectiveness of our approach. The results show that 32,183 lines of code are reused by writing only 1,033 lines of code including the compatibility layer. This approach saved about 97% of the coding work for supporting Mindstorms exclusive devices in EV3RT.

2.6.3 Bluetooth Support with BTstack

The middleware of Bluetooth protocol stack is needed in order to support Bluetooth communication in EV3RT. BlueZ, the protocol stack for Linux, is hard to be used in EV3RT as explained above. Instead, we integrated BTstack [21], an open source Bluetooth protocol stack for embedded systems into EV3RT.

Since BTstack can even work without an OS, we believe it can be ported to our platform easily. The architecture of BTstack is shown in Fig. 2.13. We implemented the hardware initialization for the Bluetooth chipset by referencing the Linux device driver. The chipset and BTstack are communicating with each other via a UART port. We implemented the method stubs defined in the UART hardware abstraction layer of BTstack. BTstack uses a run loop to handle incoming data and schedule work. By default, BTstack only provides two types of run loops, one for POSIX system and the other one for OS-less system. The run loop for OS-less system is executed as a busy loop. We modified it to run as a task in the core services layer. A QoS task is used to dynamically control the priority of the run loop task, as described in Section 2.3.5. At last, we implemented the packet handlers (PH) to provide services such as Bluetooth Serial Port Profile (SPP) which can emulate a serial port wirelessly.

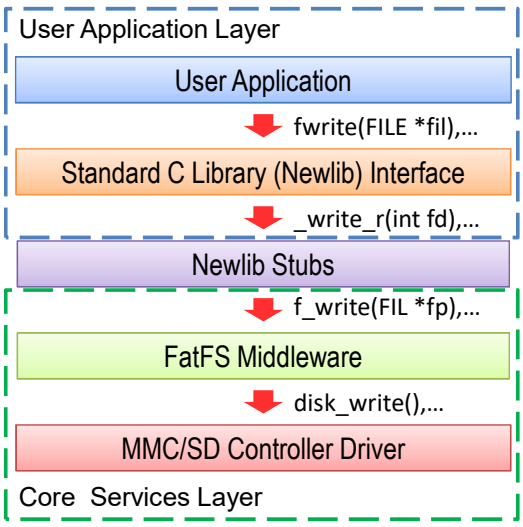

Fig. 2.14 Call flow of a standard file operation

EV3RT hides BTstack from user applications since developers have to understand the Bluetooth protocol, which might be very difficult, in order to use it. An API to open the RFCOMM serial channel as a file is provided. With that file obtained, developers can easily do communications by using functions such as `fprintf()` and `fgetc()`. The serial I/O interface in HRP2 kernel is also supported. For firmware or platform developers using standalone mode, BTstack API can be used directly.

2.6.4 File System Support with FatFS

File system support is also very important since many applications fetch and store data from/to files in the microSD card. The bootloader of EV3 requires the microSD card using FAT file system. Therefore, we integrated FatFS [29], a generic FAT file system middleware, into EV3RT. FatFs is completely written in ANSI C (C89) and has no platform dependence. It requires the implementation of a hardware abstract layer, called media access interface, which only includes six functions to access the physical devices. We implemented the media access interface with the MMC/SD controller driver to support the access of microSD card.

However, as FatFS has its own platform-independent API, it cannot cooperate with the standard C library directly. That is to say, user cannot use functions like `fscanf()` with a FatFS file. We created a bridge between Newlib and FatFS, which associates every Newlib file descriptor with a FatFS file, to overcome this limitation. The call flow of a standard file I/O function is shown in Fig. 2.14. Thus, to access files in the microSD card, users do not

Table 2.3 Basic characteristics of EV3RT, leJOS and MonoBrick

	EV3RT	leJOS	MonoBrick
Boot time	2.0s	65.5s	105.4s
Memory footprint	2,230 KiB (3.4%)	58,888 KiB (89.9%)	43,224 KiB (66.0%)
CPU utilization	1%	6%	5%

have to learn the usage of FatFS at all. In fact, they are not allowed to use FatFS directly when developing in dynamic loading mode, in order to avoid conflicts.

2.7 Performance Evaluation

Performance of EV3RT is evaluated and compared with two existing software platforms, leJOS [123] and MonoBrick [120], in this section. leJOS supports developing applications for EV3 in Java while MonoBrick allows users to program in C# with an open source .NET framework called Mono. Both of them are based on Linux kernel and use virtual machine as runtime to execute applications. EV3RT version Beta 6-2, leJOS EV3 version 0.9.1, and MonoBrick with firmware version 1.2.0.39486 are used in the comparison. According to the results, dynamic loading mode and standalone mode in EV3RT have almost the same execution performance. We only show the performance under dynamic loading mode in this section since it is recommended for developing user applications.

At first, we measured some basic but important characteristics, including boot time, memory footprint and CPU utilization, of these platforms. The amount of used memory and CPU usage is acquired using `top` command in leJOS and MonoBrick. In EV3RT, memory footprint is known at compile time, and CPU usage can be calculated by monitoring task dispatcher. The results are shown in Table 2.3. EV3RT can boot over 30 times faster than others, owing mainly to the static design of whole platform. It only uses about 2MB memory (3.4% of the whole available RAM) for the base system. The very small memory footprint of EV3RT allows developers to store and process bigger data sets in their application. All these platforms consume very little CPU resource and EV3RT is the best of them.

We then used HaWe Brickbench [56] version 1.09.2, a benchmark test for Mindstorms software platforms, to evaluate the overall performance. The benchmark has been performed for 1,000 times and the execution time of each test in milliseconds are shown in Table 2.4. EV3RT shows much more stable results and it also outperforms leJOS and MonoBrick in most cases. The average performance of LCD graphics on EV3RT is a little worse than MonoBrick because we have not optimized the implementation of graphics.

Table 2.4 Execution time of HaWe Brickbench benchmark in ms

	EV3RT			leJOS			MonoBrick		
	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.
Integer $+-$	2	2	3	1	2	151	1	1	19
Integer $\times\div$	12	12	13	9	10	69	25	28	50
Float math	26	26	26	40	43	174	191	209	275
Random number	1	1	3	4	5	79	12	12	24
Matrix algebra	5	5	6	28	33	265	39	43	286
Array sort	72	74	78	80	86	255	141	154	279
LCD text	56	56	58	155	163	356	181	217	346
LCD graphics	178	178	180	305	312	661	144	164	493

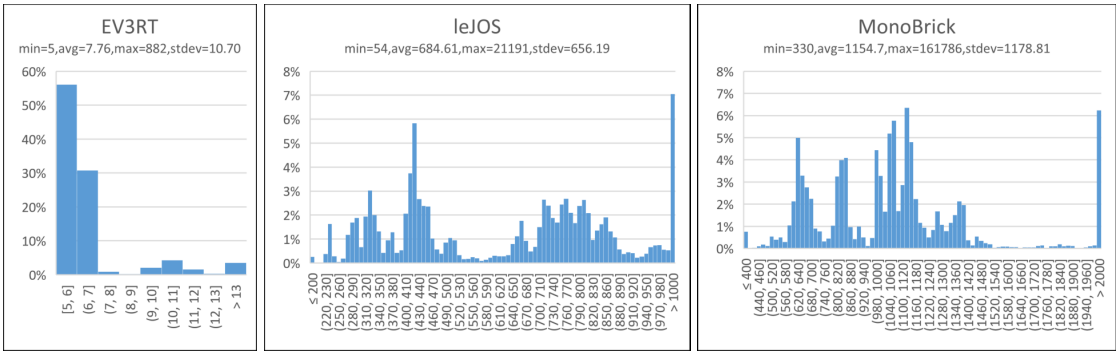

Fig. 2.15 Histograms of observed thread switching latency in μs

HaWe Brickbench only tests the computational performance of intelligent brick. In order to evaluate the performance in real-time robot control, we also measured thread switching, motor control and sensor access on these platforms. The thread switching latency is critical in multi-threaded robot control. It is also very important in single thread program with real-time requirements, since there are always some system threads running in the background. Overhead of motor control is measured by calling API to set power of a servomotor, and sensor access is measured by calling API to read distance from an ultrasonic sensor. We measured them for 10,000 times on each platform, and the results are shown in Fig. 2.15, Fig. 2.16 and Fig. 2.17. Averagely, EV3RT is about 100 times faster in thread switching, 40 times faster in motor control and 10 times faster in sensor access. Further, its performance is much more predictable than others, which makes it the most suitable platform for real-time applications.

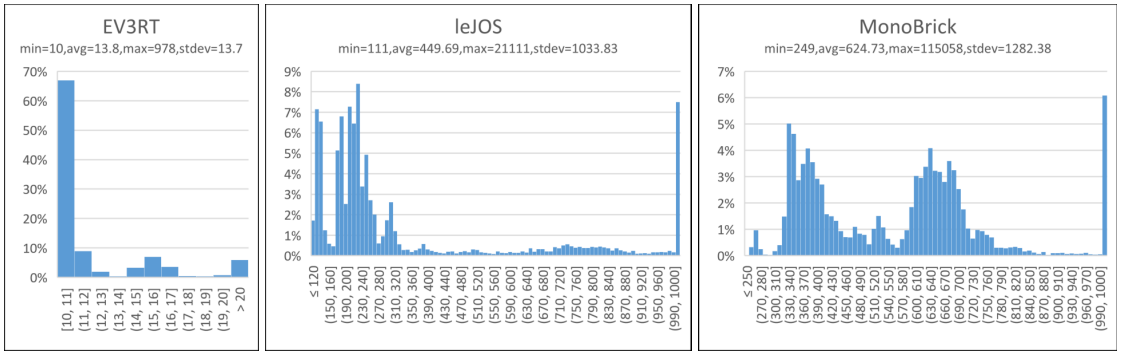

Fig. 2.16 Histograms of motor control API's overhead in μs

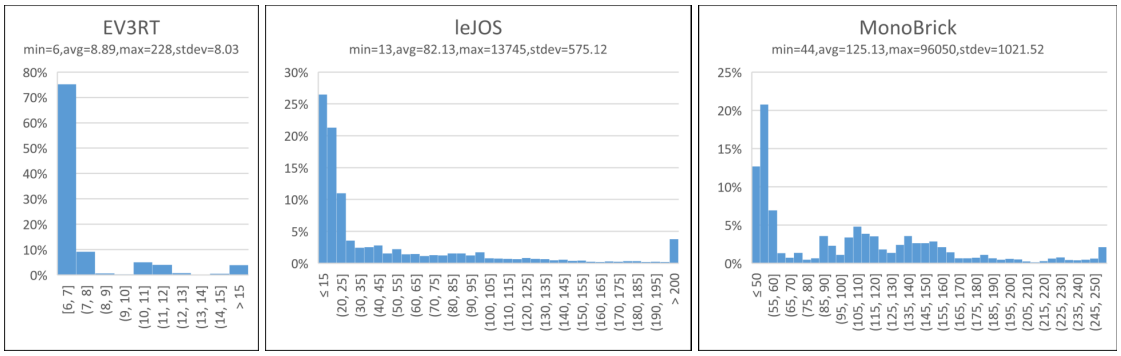

Fig. 2.17 Histograms of sensor access API's overhead in μs

2.8 Conclusions

In this chapter, EV3RT, a novel real-time software platform for LEGO Mindstorms EV3 robotics development kits, has been presented. It is a developer-friendly open source platform with an extendable layered architecture and easy-to-use APIs. Two development modes are supported: dynamic loading mode for user application, and standalone mode for firmware and platform. Sample applications to show the usage of EV3RT are also described. The protection functionalities of HRP2 kernel are applied to make EV3RT a reliable platform. A dynamic module loading mechanism for static OS has been proposed. It has a hardware-independent design and does not require dynamic memory management. By following this mechanism, application loader for EV3RT is implemented to provide a smooth development process. The development cost of EV3RT itself are sucessfully saved by reusing existing open source software. EV3RT can startup much faster, consume less resources and is the most suitable software platform for developing application with high real-time requirements, compared to existing Linux-based platforms.

Chapter 3

FMP-MC: Analysis and Optimization of RTOS Scalability for Many-Core

3.1 Introduction

Multi-core processors have been used in many embedded systems to satisfy the increasing performance requirements with a reasonable power consumption. Most of those embedded systems are based on a multi-core RTOS because they usually include applications with real-time constraints. Previous studies of multi-core RTOS have been mainly focused on the schedulability and response time analysis of task sets on multi-core processors [108, 133]. Meanwhile, many researchers have claimed that high-end embedded systems in the near future will also include high-performance parallel applications in order to support complex tasks like autonomous driving of vehicles [98].

Current mainstream embedded multi-core processors are not suitable for those parallel applications since they only have a small number of cores. Several off-the-shelf many-core processors aiming for future embedded systems, which contain tens (or even hundreds) of cores, have been released in recent years. Examples of those processors include the 72-core Mellanox TILE-Gx72 processor [88] and the 288-core Kalray MPPA (Multi-Purpose Processor Array) [61]. However, it remains unclear whether traditional RTOS can allow parallel applications to scale well on many-core processors, since they are not designed to provide high scalability for these applications.

In the field of high-performance computing (HPC) and cloud computing, on the other hand, the scalability problems in traditional general-purpose operating system (GPOS) have been actively researched for decades. Linux, the de facto standard kernel for HPC and cloud servers, has been considered as a bottleneck for processors with a huge number (hundreds to

thousands) of cores. Some OS kernels specially designed to avoid scalability bottlenecks, such as Barrelfish[15], Corey [23], and fos [138], have been proposed. However, these kernels with new designs require different methodologies for implementing user applications (e.g. explicitly control sharing), which can make the development much more complicated than the traditional approach. Meanwhile, researchers have also shown that Linux can actually provide good scalability on many-core processors with tens of cores (or at least on a 48-core machine) and thus "there is no scalability reason to give up on traditional operating system organizations just yet" [24].

In this chapter, we focus on the scalability of traditional multi-core RTOS on many-core processors with less than 100 cores. We believe those processors are very likely to become the mainstream embedded processors in the near future. The analysis is conducted by comparing RTOS-based runtime system with the well-optimized Linux-based one. The word "runtime system" means the OS kernel and all necessary middleware required by the user application. If RTOS shows close (or better) scalability comparing to Linux in parallel benchmark, we can say that traditional RTOSes are, at least potentially, suitable for scaling parallel applications on embedded many-core processors.

The main contributions of this research are as follows. Firstly, *FMP-MC*, a prototype platform for many-core embedded systems is presented. It is based on a traditional multi-core RTOS kernel called TOPPERS/FMP [126] and supports the 72-core TILE-Gx72 embedded many-core processor [88]. It provides necessary runtime system to execute parallel applications. We believe that it is the first publicly released open source testbed for evaluating traditional RTOS on an off-the-shelf many-core processor [76]. The second contribution is that several bottlenecks in RTOS have been identified by comparing with Linux. The methods to avoid them are then discussed. At last, the PARSEC benchmark suite [95] is used to evaluate and analyze the scalability of RTOS and Linux. To our knowledge, no previous studies have compared the performance of traditional RTOS and Linux on a real many-core processor. The results suggest that traditional RTOS, after proper optimization, tends to deliver better performance than Linux, and thus it is still a good choice for embedded many-core systems in the near future.

The rest of this chapter is organized as follows. In Section 3.2, we overviews the experiment environment of this research. In Section 3.3, bottlenecks in RTOS-based runtime system are addressed by comparing with Linux. The performance evaluation using PARSEC is described in Section 3.4. Finally, this research is concluded in Section 3.5.

3.2 Experiment Environment Overview

The biggest obstacle before going any further into the analysis is the lack of an open source testbed that allows us to evaluate a traditional RTOS on a many-core processor with high-performance parallel applications. Therefore, we decided to build an experiment environment from the ground up at first.

TILEncore-Gx72 [89] is used as the hardware of our experiment environment. It is an off-the-shelf development board equipped with a 72-core embedded many-core processor called TILE-Gx72 and 32 GByte DDR3 memory. TOPPERS/FMP, a traditional multi-core RTOS kernel, is ported to TILE-Gx72. FMP-MC, a prototype software platform with runtime system to execute parallel applications, is then built on TOPPERS/FMP. The source code of FMP-MC has been made publicly available [76]. PARSEC [95], a popular benchmark suite composed of multithreaded applications, is used for evaluating the performance and scalability. Experiments are performed both on FMP-MC and a Linux 4.5 based runtime system for comparison analysis.

In this section, three most basic elements in our experiment environment—the TILE-Gx72 processor, the TOPPERS/FMP kernel and the PARSEC benchmark suite—are overviewed. Details like the runtime system of FMP-MC will be discussed in later sections.

3.2.1 TILE-Gx72 Embedded Many-Core Processor

TILE-Gx72 is a 72-core processor from the TILE-Gx many-core processor family which aims for delivering high performance and energy efficiency to embedded systems and servers [88, 89]. Cores (called tiles) in TILE-Gx72 are interconnected with a 2D mesh NoC (network-on-chip) named iMesh as shown in Fig. 3.1. iMesh uses a dimension-ordered (X-Y) routing algorithm and the latency is 1 clock cycle per hop. Each core is a full-featured, 64-bit processing unit working at 1 GHz, including 32 KByte private L1 instruction cache, 32 KByte private L1 data cache, 256 KByte L2 cache, and a full-blown memory management unit (MMU). There are 4 memory controllers and each of them is directly connected to a single core. Although other cores need to use iMesh network to access memory, the process is transparent to system developers.

TILE-Gx72 can provide hardware cache coherence by a technology called dynamic distributed cache (DDC). The basic idea of DDC is to use the union of all of the L2 caches as a distributed virtual L3 cache. Each physical memory address in TILE-Gx72 is associated with a home tile, which is set in the corresponding page table entry. The coherence information for a particular physical address is always tracked and maintained by its home tile. If a core wants to cache an address homed remotely into its local L2, it requests the data from the

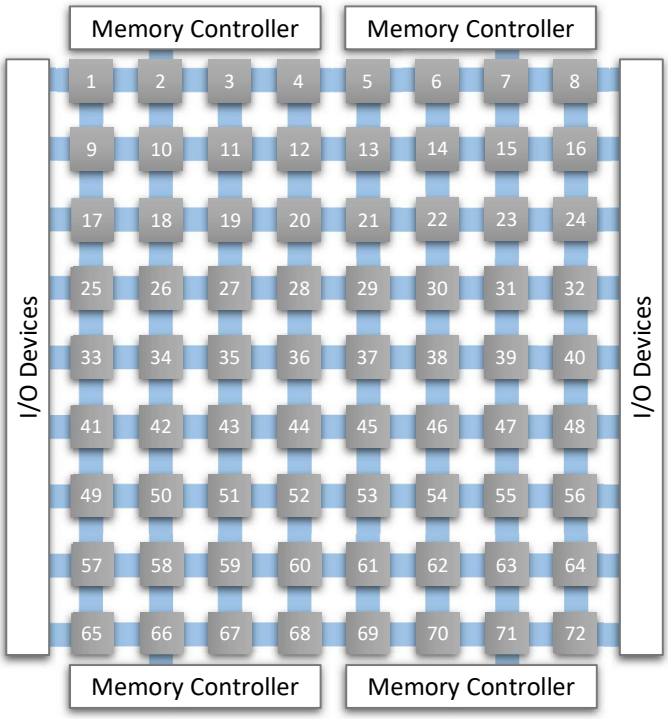

Fig. 3.1 iMesh NoC in TILE-Gx72 processor

home tile instead of the memory controller. Likewise, if a core modified a L2 cache line, it will also send the new data to the home tile and the home tile will invalidate all the other copies for consistency. In this way, the L2 cache in a home tile can be viewed as a coherent L3 cache, and thus the TILE-Gx72 can be logically treated as a traditional shared memory system despite its NoC architecture.

Since a memory page is much larger than a cache line (e.g. small page is 64 KByte and L2 cache line is 256 Byte), it could by very inefficient to home an entire page with a single core in some cases. TILE-Gx72 provides a strategy called hash-for-home to maximize the average performance by effectively utilizing L2 cache and NoC bandwidth of the entire chip. If a page is marked as hash-for-home, its addresses will be distributedly homed across multiple cores (e.g. 1st L2 cache line by core 1, 2nd L2 cache line by core 2, ...). The hash-for-home strategy is recommended as the default homing policy because it can generally deliver excellent throughput.

3.2.2 TOPPERS/FMP Multi-Core RTOS Kernel

TOPPERS/FMP (or "FMP" for short) [126] is an open-source traditional multi-core RTOS kernel based on the μITRON specification [132]. FMP is short for Flexible Multiprocessor

Profile, and as its name implies, FMP has a flexible implementation which can support both symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP) architectures [125].

Only the most common basic functionalities such as multitasking and time management are provided in FMP itself, since the services required by different embedded systems can vary widely. Advanced features (e.g. file system or networking) are supported by optional middleware implemented with the APIs of FMP. Therefore, when evaluating the performance of applications, we must take the whole runtime system into consideration.

FMP uses a static system design which will generate all kernel objects during compile time. Although the static design is less flexible than the dynamic design in GPOS kernel, it tends to have lower overhead because of its simplicity.

Like most multi-core RTOS kernels, the target systems of FMP are embedded systems with a small number (usually less than 10) of cores. In the original implementation of FMP, the maximum number of cores are dependent on the word length of the processor. Although it is enough for current mainstream processors, for the 64-bit TILE-Gx72 processor, only 64 of the 72 cores can be used by the kernel. We have investigated into this limitation and found it can be easily fixed. The original data type of processor affinity mask, which determines the set of cores a task can run on, is `uint_t` whose size is equal to the word length. Each bit in the mask represents a single core, and hence the number of cores is limited. We have modified the source code to use `uint32_t[`$\lceil \frac{CoreNumber}{32} \rceil$`]` for these masks. Because the new data type can scale with the actual core number, FMP in our experiment environment can, at least theoretically, support many-core processors with an arbitrary number of cores.

3.2.3 PARSEC Benchmark Suite

The Princeton Application Repository for Shared-Memory Computers (PARSEC), is a benchmark suite for evaluating the performance of shared-memory Chip-Multiprocessors (CMPs) [95]. TILE-Gx72 can be viewed as a shared-memory CMP because all cores can coherently access the same memory system via iMesh. The most important reason for choosing PARSEC is that it focuses on next-generation applications for future CMPs [20]. Key characteristics of PARSEC are listed as follows.

Multithreaded. All applications in PARSEC have been parallelized to utilize resources of multi-core processor as possible. Various programming models such as fork-join and pipeline are covered for analyzing from different perspectives.

Emerging. PARSEC includes workloads which are considered important in the near future but not commonly used yet. It can be very helpful for figuring out to what extent a processor can meet the demands of emerging applications.

Diverse. PARSEC does not focus on some specific application domain (e.g. HPC). Instead, it includes a wide spectrum of applications, such as those for desktop and servers. Owing to this diversity, PARSEC tends to trigger more performance bottlenecks than those domain-specific benchmarks.

PARSEC 3.0 is used to evaluate our experiment environment. The original build system provided by PARSEC creates applications as executable files (e.g. ELF file in Linux) to be executed by the OS. As a static RTOS, FMP does not have a loader to execute application file at run time. We have extended the build system of PARSEC to allow an application to be created as a static library. The generated library file of PARSEC application can be linked together with the RTOS to form a single image which can be directly executed by the boot loader.

3.3 Runtime System Analysis and Optimization

Most user applications cannot directly run on OS kernels. Instead, runtime environments (RTEs) built above the kernels, which include necessary middleware and libraries, are required to execute them.

In this section, the characteristics of the runtime systems for FMP and Linux are analyzed. Some performance problems in the FMP-based runtime system (i.e. FMP-MC) have been identified and our solution for each problem is explained and evaluated.

3.3.1 OS Kernels

FMP and Linux have many differences in kernel design and implementation because they are not targeting the same systems. In this section, we will focus on comparing the factors which can have impact on the performance of applications.

3.3.1.1 System Design: Static vs. Dynamic

The primary function of an OS kernel is to manage hardware and software resources in the system and provide services to access them. This function can be implemented either in a static or a dynamic approach, depending on the target systems.

For most embedded systems, the required and available resources are predetermined since these systems are designed to perform some specific tasks. In FMP, kernel objects (e.g. tasks, semaphores, data queues) are statically configured, by defining them in configuration files at design time. All necessary data structures for these objects, such as control blocks,

will be generated by a configurator during compile time. Each kernel object is associated with an ID for accessing with system calls.

Linux, on the other hand, has a dynamic design in order to provide the flexibility demanded by general purpose computer systems. Kernel objects are created and initialized at run time with Linux in-kernel APIs such as `kthread_create()`. Usually, these creation APIs will return a pointer of the data structure dynamically allocated for that object, which is needed for accessing.

Since the number of kernel objects in FMP is fixed at run time, they can be simply stored in arrays and referenced by using index as ID. Meanwhile, data structures of kernel objects in Linux are managed using linked lists of dynamically allocated memory blocks, because their number is considered to be unbounded. This kind of complexity introduced by a dynamic design can lead to higher overhead compared to the straightforward implementation of a static design. Previous studies have also shown that static kernel, while has limited flexibility, can achieve smaller footprint, better real-time performance and greater reliability [77, 58].

3.3.1.2 Basic Services and Device Drivers

From the viewpoint of application developers, an OS kernel consists mainly of basic services and device drivers, whose functionalities are usually provided as system calls.

Basic services are minimal, yet essential, elements which can be found in most OS kernels. Typical examples include memory management, time management, interrupt handling, kernel locking, and multitasking with primitives for synchronization and communication. Although both FMP and Linux support these functions, the implementations differ due to their system designs. Generally, accessing services in FMP has lower overhead while Linux is optimized to achieve better throughput and scalability.

The duties of device drivers, meanwhile, are not alike inside FMP and Linux. Since hardware of an embedded system is highly specialized for certain application domain, it is difficult for the kernel to assume what device drivers should be provided. FMP itself only includes the minimum necessary device drivers, such as those for timer, interrupt controller and serial port (if syslog is used). Other devices are supported by optional middleware components implemented on top of the kernel which are considered to be part of the RTE. Developer is required to use dedicated APIs of each middleware to control corresponding device.

On the contrary, Linux, as a GPOS kernel, must support most of the common devices, such as networking and file systems, out of the box and provide a universal approach to access them. Linux has a system call interface which has been kept stable over the decades in order to ensure portability of applications. Drivers are modules inside the kernel and

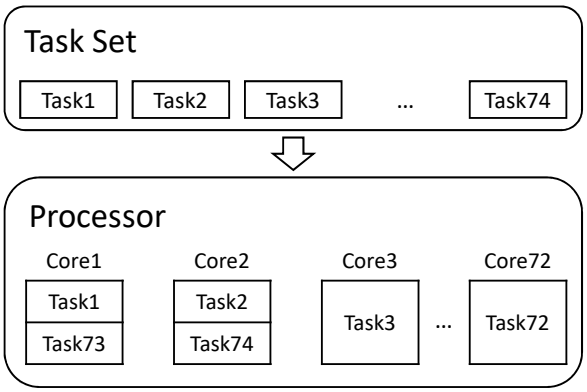

Fig. 3.2 Example of mapping tasks in a round-robin fashion

interact with applications via that system call interface. More specifically, devices in Linux are abstracted as special files called device files and can be accessed with file I/O system calls like `write()`. Although this mechanism in Linux unifies interaction of device drivers using standard file operations, it also introduces overhead in the file system.

3.3.1.3 Task Scheduling Disciplines

Scheduler is one of the most important module in a kernel to support multitasking. Its discipline can greatly influence the scalability of an application in some cases. The objective of scheduling in multi-core systems is to determine how tasks are mapped to different cores and how CPU resources are allocated to tasks on each core, in order to meet the goals like maximizing throughput or minimizing response time. Since it is an active area of research and still has many challenges ahead [115], we will not focus on state-of-the-art methodologies in this research. Instead, some simple but effective disciplines are used for evaluation.

In FMP, tasks in user applications are scheduled by a method called Round-robin then FIFO (RtFIFO). Tasks are assigned to cores in a round-robin fashion at creation. Fig. 3.2 shows an example of how 74 tasks are assigned to the TILE-Gx72 processor. The core indexes in this figure correspond to those in Fig. 3.1. Each core will get a task in turn until all tasks have been assigned. Tasks on the same core (e.g. task 1 and task 73 in the example) will be handled by the default fixed-priority preemptive scheduler. All tasks are set to the same priority so they will be served in FIFO fashion on each core. RtFIFO is a straightforward scheduling discipline without any dynamic load balancing mechanism. It is very effective for those parallel applications using the fork-join model with no load imbalance problems.

Linux, by default, uses the Completely Fair Scheduler (CFS) which aims to maximize overall CPU utilization. CFS periodically runs a complex load-balancing algorithm to keep the runqueues of all cores roughly balanced. It can improve the CPU utilization for

applications using the pipeline model which often has load imbalance issue [91]. However, thread migrations between cores can be extremely expensive and may significantly hurt cache locality in some cases [84].

Since it is not fair to compare two runtime systems with different scheduling disciplines, we have also implemented RtFIFO on Linux. The `pthread_attr_setaffinity_np()` function is used to bind a thread to a specific core. Threads are set to the `SCHED_FIFO` policy so they will be handled by the fixed-priority preemptive scheduler [63].

Traditionally, task assignment policy in distributed shared memory (DSM) and cache coherent NUMA (ccNUMA) systems must also consider the cost of memory access from different core. Interconnects in those systems have limited bandwidth and high latency compared to on-chip communication, which can often become a bottleneck. On the other hand, the latency of iMesh on-chip network is only 1 cycle per hop and the bandwidth of each core is over 1 terabit/s. The fast NoC allows TILE-Gx72 processor to provide the hash-for-home memory homing policy, as described in Section 3.2.1, which can distribute memory access across all cores in granularity of L2 cache line size. Since this policy can achieve great throughput in general and is unlikely to be a bottleneck in average case, the distance between core and memory controller is not considered by schedulers for TILE-Gx72 processor. Instead, for situations with special memory access pattern, the throughput could be further improved by controlling the home of a memory page, such as the optimization in Section 3.3.1.4.

3.3.1.4 Memory Management

Memory management is supported by most OS kernels, including static OS kernels like FMP, since there are many applications requiring the flexibility of dynamic resource management. Its function mainly consists of two parts: management of objects in kernel and management of data in user applications.

For kernel objects, the mechanisms used by FMP and Linux to manage them are quite similar. Although FMP does not support dynamic allocation of memory, it does allow user to statically define objects like memory pools for fixed-size blocks with predetermined maximum number. Therefore, dynamic management of kernel objects can be easily implemented by using the object pool design pattern [113]. In Linux, a mechanism called slab allocation [22], which is also based on object pools, is used for the efficient memory allocation of kernel objects.

For processor like TILE-Gx72 which uses paged MMU, dynamic allocation of data for user applications technically means associating page table entries with free physical memory. In FMP, all page tables are statically generated with unused physical memory defined as

a pool which can be accessed from user application. That is to say, from the viewpoint of pages, all free physical memory has already been allocated to user application in advance. The preallocated pool of free memory can be logically managed by a memory allocator middleware and there is no need to dynamically change page tables at run time. Meanwhile, Linux has an extremely complex implementation for page table management with advanced features including demand paging and swapping [106]. Modifying page tables dynamically could be a bottleneck in some cases and has been actively studied and optimized over many years [34]. Consequently, if an application frequently requests the memory management services from kernel, it may have better performance on FMP than Linux.

As mentioned in Section 3.2.1, hash-for-home strategy for pages can provide higher throughput in general. However, there do exist some situations where homing an entire page with a single core is likely to deliver better performance due to data locality. For example, the stack of a task is usually considered as a private memory area which will be heavily used by its owner but barely be accessed from other tasks. A task will have to frequently communicate with other cores via iMesh if its stack is distributedly homed by hash-for-home strategy. Memory manager in Linux includes an optimization which will home data structures like stacks locally, in order to improve performance for these cases.

We have also introduced a similar optimization to the process of page table generation in FMP. Firstly, each core has its own section (e.g. "`.local_cached_prc1`" for core 1) to store local data. While generating source code for kernel objects, data structures such as task stacks, interrupt stacks, task control blocks and processor control blocks will be placed into the section of its local core, using the section attribute supported by the compiler. At last, page table entries for each section will be generated with home set to the corresponding core. The effectiveness of this optimization is difficult to quantify since it is heavily dependent on how these data structures are specifically used by application. An example of its effect on spinlocks will be shown in Section 3.3.1.5.

3.3.1.5 Kernel Locking

Locking is an essential mechanism in multi-core OS kernels to support inter-core synchronization and communication. The granularity of locking model in a kernel has a great impact on the overall throughput, since almost all the shared resources are required to be protected by locks.

In FMP, there are three levels of granularity to choose from: giant lock (`G_KLOCK`), processor lock (`P_KLOCK`) and fine-grained lock (`F_KLOCK`). In giant lock mode, all kernel objects share a solitary global lock and thus kernel services like system calls can only be accessed serially. This mode requires least memory space but has highest resource contention.

In processor lock mode, kernel objects on the same core share a single lock and thus requests of kernel services on different cores can be processed concurrently. In fine-grained lock mode, each kernel object has its own lock so kernel services can be provided as parallel as possible. This mode has the lowest resource contention but will use much more memory than other modes if there are a large number of kernel objects. Since our experiment environment has lots of available memory, we uses the fine-grained lock mode to maximize parallelism.

In previous versions of Linux, there used to be a Big Kernel Lock (BKL) which is just like the giant lock in FMP. From Linux 2.6.39, the BKL has been completely removed and replaced by a fine-grained locking scheme [83]. The granularity of locking in current Linux kernel is a bit coarser than the fine-grained lock mode in FMP because Linux is more complex and includes many components consisting of multiple kernel objects.

Spinlock is the most basic primitive for locking and advanced locks like semaphores are implemented using spinlock APIs. Therefore, spinlock implementation in kernel can heavily influence the scalability of the whole system. Typically, test-and-set (TAS) spinlock and one of its optimized variant called test-and-test-and-test (TATAS) spinlock [6] are used in RTOS for embedded systems like FMP. TAS and TATAS spinlocks have extremely simple implementations (usually several lines of C code) and only require the hardware to support a single atomic operation—the test-and-set instruction—which can be found in most multi-core processors. Although they can provide excellent performance on embedded systems with only a few of cores due to the simplicity, the throughput can dramatically collapse on many-core processors because the implementations are not scalable [25]. In Linux, a ticket spinlock with exponential backoff (hereinafter "BACKOFF spinlock") is used for TILE-Gx72 processor. The BACKOFF spinlock is not scalable either but considered to have better throughput than TAS and TATAS spinlocks [6].

In order to evaluate the scalability of different spinlocks on TILE-Gx72, we have implemented BACKOFF spinlock in FMP and TAS and TATAS spinlocks in Linux. We have also implemented K42 spinlock [12], which is a variant of the scalable MCS spinlock with compatible APIs, to compare the difference between scalable and non-scalable implementations.

We have measured the throughput of different spinlocks on FMP and Linux with a microbenchmark, and the result is shown in Fig. 3.3. In our microbenchmark, each core will loop for 100,000 times to acquire a shared spinlock, read and write 4 shared cache lines, and then release the lock. The throughputs of TAS and TATAS spinlocks are indeed better than others when the core number is small (less than 6) but will decrease rapidly. BACKOFF spinlocks have a much slower speed of throughput decreasing than TAS and TATAS and can still provide a relatively acceptable performance when all cores are used. K42 spinlocks can maintain good scalability as core number increases, but have the worst performance when

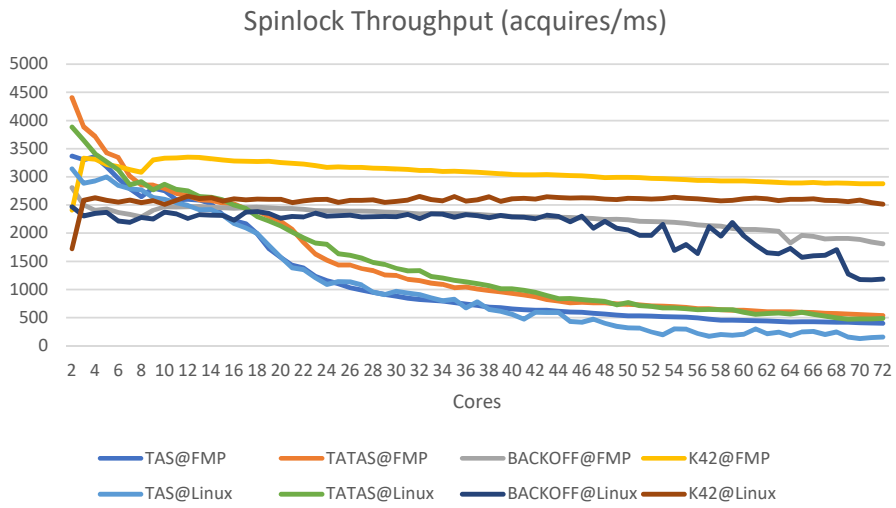

Fig. 3.3 Spinlock throughput of different implementations

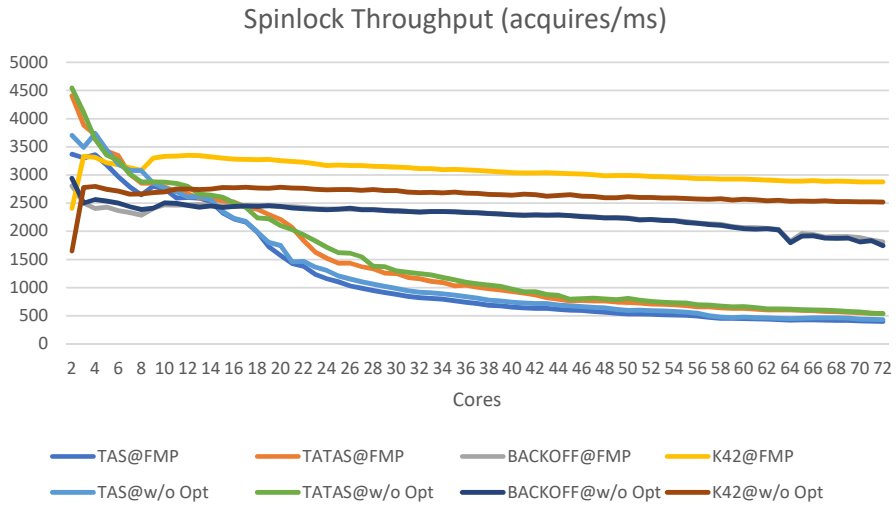

Fig. 3.4 Spinlock throughput in FMP with and without page optimization

less than 3 cores compete for the shared resource. Generally, FMP shows better performance than Linux when the same spinlock implementation is used. We believe that it is mainly because background system tasks in Linux can sometimes preempt the execution of our microbenchmark application.

When comparing FMP with Linux, the page table optimization introduced in Section 3.3.1.4 is always enabled for the sake of fairness. We have also measured the throughput of spinlocks for FMP when the optimization is disabled and the result is shown in Fig. 3.4. It can be seen that this optimization barely has any effect on TAS, TATAS and BACKOFF spinlocks. However, it does increase the scalability of K42 spinlocks for about 20%. It is

because that K42 is the only implementation in them which actively uses the task stacks (for spinlock queue node).

3.3.1.6 Getter Function for Thread ID

One of the easiest and most effective methods to enhance scalability is replacing a globally shared data structure with a thread-local one. Since each thread has its own copy for that variable, they do not need to use a lock for synchronization and cache line contention can also be reduced.

In order to access a thread-local variable, a thread must be aware of its identifier. Thread libraries and OSes usually provide a function to get the ID of a thread (e.g. `pthread_self()` in pthread, `gettid()` in Linux and `get_tid()` in FMP). If a program frequently accesses thread-local data, the overhead of the getter function will become important. In Linux (and its pthread library) for TILE-Gx72, a special register named `tp` is used to store the thread ID, and getter functions can just read and return the value in `tp` register with a few instructions. Meanwhile, the getter function in FMP has a generic and safe implementation. It does not depend on any detailed hardware specification, but instead, has to lock the core before reading the value from memory. Further, an error code will be returned if it is not called from the task context. Consequently, the default getter implementation in FMP has larger overhead than the optimized one in Linux.

To avoid the bottleneck caused by getter, we have optimized the getter function in FMP with two methods: the `FAST` method and the `SPR` method. The `FAST` implementation is still hardware independent but has all error checking code in the getter function removed. Developer is responsible for calling this `FAST` function properly. The `SPR` implementation is, on the other hand, using a special register just like the optimization approach in Linux. It only works on TILE-Gx72 but has the minimal overhead. An example of how different implementations can affect the throughput of a scalable middleware will be shown in Section 3.3.2.5. In further performance comparison with Linux, FMP always uses the `SPR` implementation of the getter function for thread ID.

3.3.2 Middleware in Runtime Environment

In this section, we analyze necessary middleware in FMP and Linux used to execute the PARSEC applications.

3.3.2.1 In-memory File System

Most applications in PARSEC use files for data input and output. The speed of storage devices can always be a bottleneck but it is not related to the runtime system. In order to rule out the factor of disk I/O, in-memory file systems are usually used when evaluating the performance of applications. In fact, the development board in our experiment environment does not even have any non-volatile storage such as hard disk drive.

For Linux, the in-memory `tmpfs` file system is used by default. For FMP, we use an in-memory file system middleware called `RAMfs`. The `fmpfs` and `RAMfs` have almost the same throughput since both of them are just redirecting the file I/O system calls to simple memory operations.

3.3.2.2 POSIX Thread Library

PARSEC applications use POSIX threads (pthreads) for multithreading functionality. As defined in its specification, a library for pthreads mainly consists of functions for thread management (e.g. creating, joining) and synchronization (e.g. mutex, condition variable, lock, barrier).

For Linux which is a POSIX-conformant kernel, a library called Native POSIX Thread Library (`nptl`) is used. For FMP which is not based on the POSIX standard, we have implemented a compatible layer called `POSIX4FMP` to provide pthreads support. Both `nptl` and `POSIX4FMP` are basically the wrapper for system calls. Although they are provided in libraries as part of the RTE, they are more like extension of the OS kernel. Therefore, their performance are actually dependent on the kernels.

3.3.2.3 C Standard Library

The C standard library (or "libc") is the standard library for C language which provides the most commonly-used functions as specified in the ANSI C standard. All applications in PARSEC heavily depend on it so its implementation can hugely influence the performance.

The SDK of TILE-Gx72 provides two libc implementations: Newlib [103] for RTOS and bare metal environment, and the GNU C library (glibc) [52] for Linux. Newlib and glibc are implemented with different system call interfaces, which means, unfortunately, we cannot use the same libc for FMP and Linux.

A libc mainly consists of functions for string handling, I/O operations, mathematical computations, and memory allocation. String handling functions are just trivial so Newlib and glibc have very close performance for them. I/O operation functions for files are basically wrappers of relative system calls in both Newlib and glibc. Other I/O functions like `printf()`

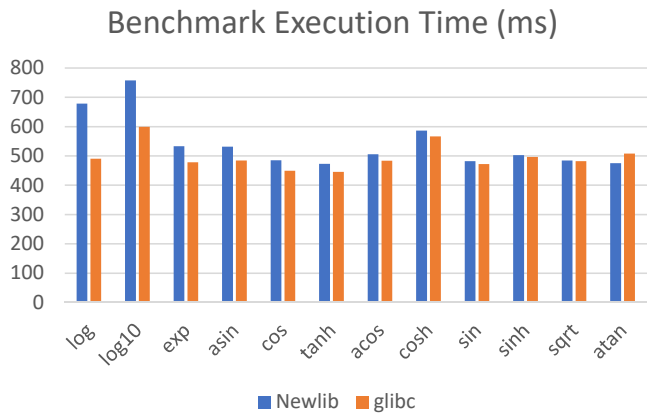

Fig. 3.5 Benchmark execution time of different mathematical libraries

are rarely used by PARSEC applications. Therefore, functions for mathematical computations and memory allocation are the most critical functions in libc which can greatly affect the performance.

In fact, both Newlib and glibc provide the mathematical functions as a separate library called `libm`. These libraries barely depend on system calls and thus we can effortlessly use them in both FMP and Linux. The mechanism to replace the functions of memory allocation is also provided by Newlib and glibc. If a library of memory allocator is linked before libc, the default memory allocator inside libc will just be overridden.

In performance evaluation, we use the same `libm` and memory allocator for FMP and Linux, which will be described later. Hence, the influence by different libc libraries is very small.

3.3.2.4 Mathematical Library

The mathematical library (`libm`), as mentioned above, can have an impact on the performance, especially for those compute-intensive applications.

Usually, RTOS like FMP just uses the `libm` in Newlib because Newlib is basically the only libc targeting for embedded systems. However, we have found that although glibc only works on Linux, the `libm` in glibc can actually be easily used in FMP.

We have measured the performance of mathematical functions in Newlib and glibc with a microbenchmark, and the result of some functions with relatively large difference is shown in Fig. 3.5. For logarithm functions `log()` and `log10()`, glibc is about 30% faster than Newlib. The `atan()` in glibc is about 6% slower. For most other functions, glibc shows slightly ($\approx$5% on average) better performance than Newlib. If an application frequently calls functions like `log()`, the `libm` in Newlib which is currently used by most embedded systems, can become a bottleneck.

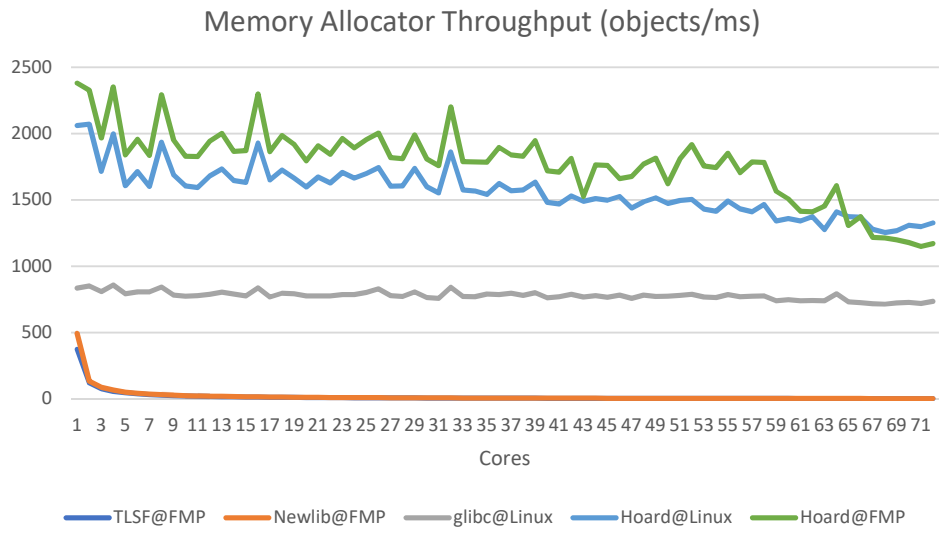

Fig. 3.6 Memory allocator throughput of different implementations

3.3.2.5 Scalable Memory Allocator

The performance of memory allocator has been studied for decades [69, 85] because it can be very important for applications requiring dynamic memory management.

In current RTOS-based embedded systems, usually two memory allocators are used: the default memory allocator in Newlib and the TLSF memory allocator [85]. The allocator in Newlib is an implementation of Doug Lea's Malloc (dlmalloc) [70] which aims for maximizing average performance and minimizing fragmentation. TLSF, meanwhile, has higher fragmentation but can guarantee bounded response time which is essential for hard real-time applications. However, both TLSF and Newlib use global locking for thread safety and thus they are not scalable.

In Linux, glibc has already used a scalable memory allocator called ptmalloc by default. Since ptmalloc is implemented with Linux API in mind, we cannot use it in FMP.

To our knowledge, there is no scalable memory allocator targeting for embedded systems currently. In order to avoid the bottleneck caused by non-scalable allocator, we have ported a scalable allocator called Hoard [19] to FMP. Hoard is designed to be cross-platform but only works on Linux, Solaris, Mac OS X, and Windows out of box. It is developed in C++ while most RTOS kernels are in C language. We have created a C language interface for RTOSes to use Hoard. Our interface only requires 8 functions, which can be easily implemented in most RTOSes, such as acquiring or releasing a lock, to be provided.

Fig. 3.6 shows the throughput of different memory allocators. Both TLSF and Newlib will collapse quickly since they are not scalable. Allocator in glibc scales excellently but the absolute throughput is only about half as good as Hoard. Hoard shows similar scalability in

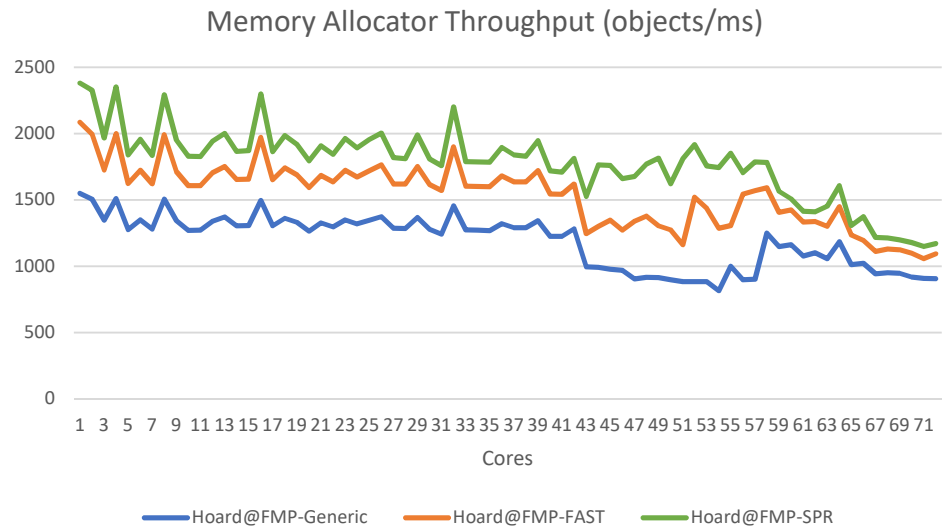

Fig. 3.7 Memory allocator throughput in FMP with different optimizations

FMP and Linux and has the best throughput. In most cases, Hoard in FMP has better absolute performance than Linux, and we believe it is because, as mentioned in Section 3.3.1.4, FMP need not to update pages at run time like Linux.

Hoard can also be a good example to evaluate the effectiveness of different optimization methods for the thread ID getter function described in Section 3.3.1.6, since it has a scalable implementation which actively uses thread-local variables. Fig. 3.7 shows the throughput of Hoard with different getter function implementations. All of them have similar trends in scalability but the generic one without any optimization shows the worst performance. The hardware-independent `FAST` optimization has about 40% better throughput than the generic. The `SPR` optimization using a special register on TILE-Gx72 has about 15% better throughput than the `FAST`. It is suggested that our two optimization methods can be very effective for scalable implementations like Hoard.

3.4 Performance Evaluation

In this section, we will evaluate and analyze the scalability of FMP and Linux using four representative PARSEC applications with different runtime system settings.

3.4.1 Runtime System Settings

Four runtime system settings, FMP-Base, Linux-Base, FMP-K42 and Linux-CFS, are used to run applications for evaluation.

Table 3.1 Runtime system settings for FMP-Base and Linux-Base

	FMP-Base	Linux-Base
Spinlock	BACKOFF	
Scheduling	RtFIFO	
File System	RAMfs	tmpfs
POSIX Thread	POSIX4FMP	nptl
C Standard Library	Newlib	glibc
Mathematical Library	libm from glibc	
Memory Allocator	Hoard	

FMP-Base and Linux-Base settings, which are shown in Table 3.1, are used to analyze the performance difference caused by different kernel implementations. For other elements in the runtime system, the same (or equivalent) conditions are used as possible. Although file systems are different, both of them are in-memory file system and, hence, have almost the same throughput. As discussed in Section 3.3.2.3, the most important functions in C standard library which can have significant impact on performance are those functions for mathematical operations and memory allocation. Since the same mathematical library and memory allocator are used, the influence of different C standard library implementations is very small for PARSEC. POSIX thread libraries are considered as extension of the OS kernel so we should always analyze them together with the kernel. Therefore, FMP-Base and Linux-Base settings are suitable for comparing different OS kernels.

FMP-K42 is the setting that replaces BACKOFF spinlock in FMP-Base with K42 spinlock. By comparing FMP-Base and FMP-K42, we can evaluate how different spinlock implementations can affect the performance. Linux-CFS is the setting that replaces RtFIFO scheduling in Linux-Base with CFS. By comparing Linux-Base and Linux-CFS, we can evaluate how different scheduling disciplines can affect the performance.

3.4.2 Blackscholes

Blackscholes is a mathematical finance application that calculates the price of options with the Black-Scholes partial differential equation. It is the simplest application in PARSEC and has negligible communication between threads. It is a data parallel application using the fork-join model.

Fig. 3.8 shows the scalability of blackscholes. Both FMP-Base and Linux-Base can scale well but FMP-Base is a bit better. K42 spinlock and CFS have negligible influence on this application.

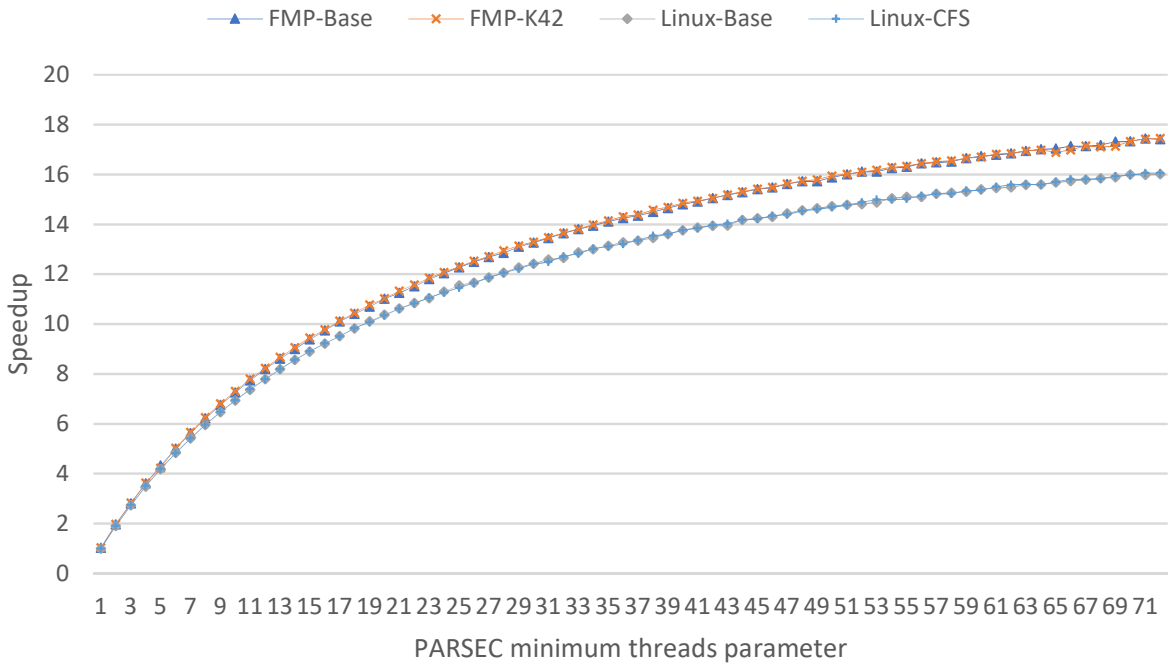

Fig. 3.8 Scalability of blackscholes with different runtime system settings

In order to analyze the reason of performance difference in FMP and Linux, we have broken down the execution time of the critical path for blackscholes. The critical path is the longest execution path of a parallel application which can determine the performance of the whole application. As a fork-join application, the critical path of blackscholes can be easily found. The execution time is broken into four parts—work for computing, sync for synchronization and communication, file for file I/O operations and allocator for memory allocation—as shown in Fig. 3.9 (with some unimportant data omitted to make the figures easy to read). We can see that work in blackscholes scales well but execution time for file does not scale at all. In fact, blackscholes handles file serially and this bottleneck has already been reported by previous study [79]. FMP can deliver better performance for blackscholes because the file operations in FMP cost less time than in Linux.

Although the in-memory file system modules in FMP and Linux have almost the same throughput, the overhead costs for accessing them are different. For example, in Linux, as described in Section 3.3.1.2, device drivers are abstracted as special files and also use the standard file operation system calls to interact with user applications. Meanwhile, those functions in FMP are only for accessing the in-memory file system. Therefore, file operations in FMP can be faster than in Linux due to the simplicity.

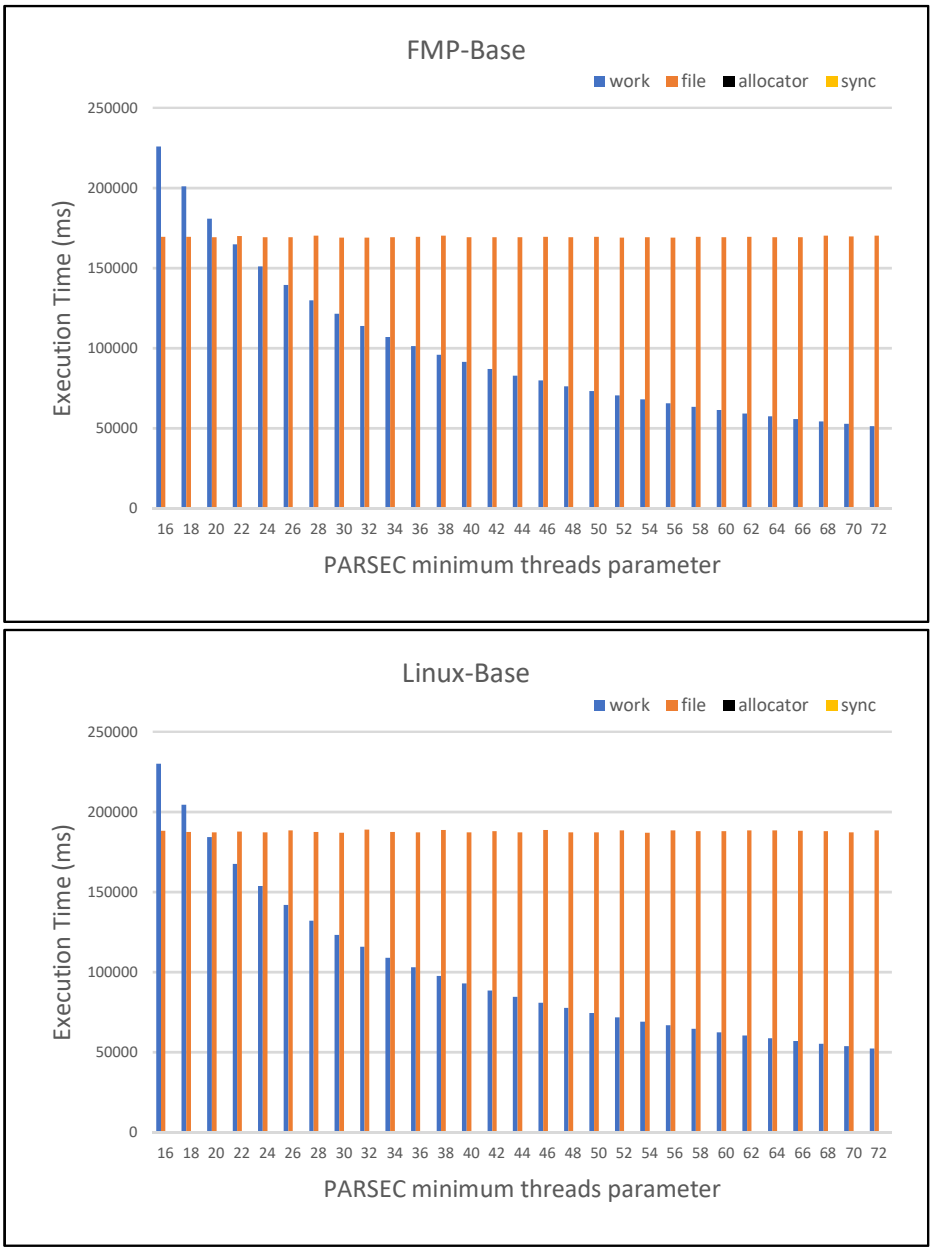

Fig. 3.9 Breakdown of blackscholes execution time

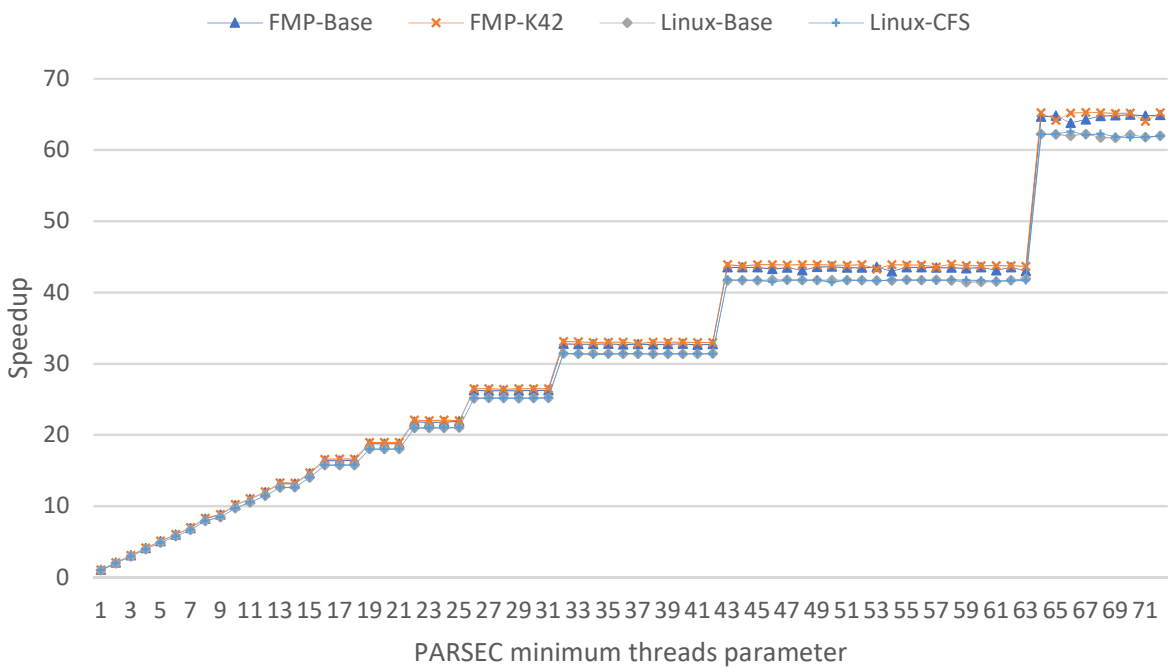

Fig. 3.10 Scalability of swaptions with different runtime system settings

3.4.3 Swaptions

Swaptions is an application which uses Monte Carlo simulation to compute the prices of swaptions. It is compute-intensive and has a little synchronization (e.g. tens of locks) between threads. It is a data parallel application using the fork-join model.

Fig. 3.10 shows the scalability of swaptions. All the runtime system settings have almost the same scalability. FMP does have a very slightly (about 5%) better performance than Linux in some cases, but the differences are too small for analysis. It is suggested that FMP and Linux have very close scalability for compute-intensive applications with little communication.

3.4.4 Streamcluster

Streamcluster is an application to solve the online clustering problem which is widely used in data mining. It has more than 100,000 barriers for synchronization between threads. It is a data parallel application using the fork-join model.

Fig. 3.11 shows the scalability of streamcluster. Linux-Base has much better scalability than FMP-Base. The scalability of Linux-CFS is very close to Linux-Base when thread parameter is less than 64, but it will decline quickly at last. The difference between FMP-Base and FMP-K42 is ignorable.

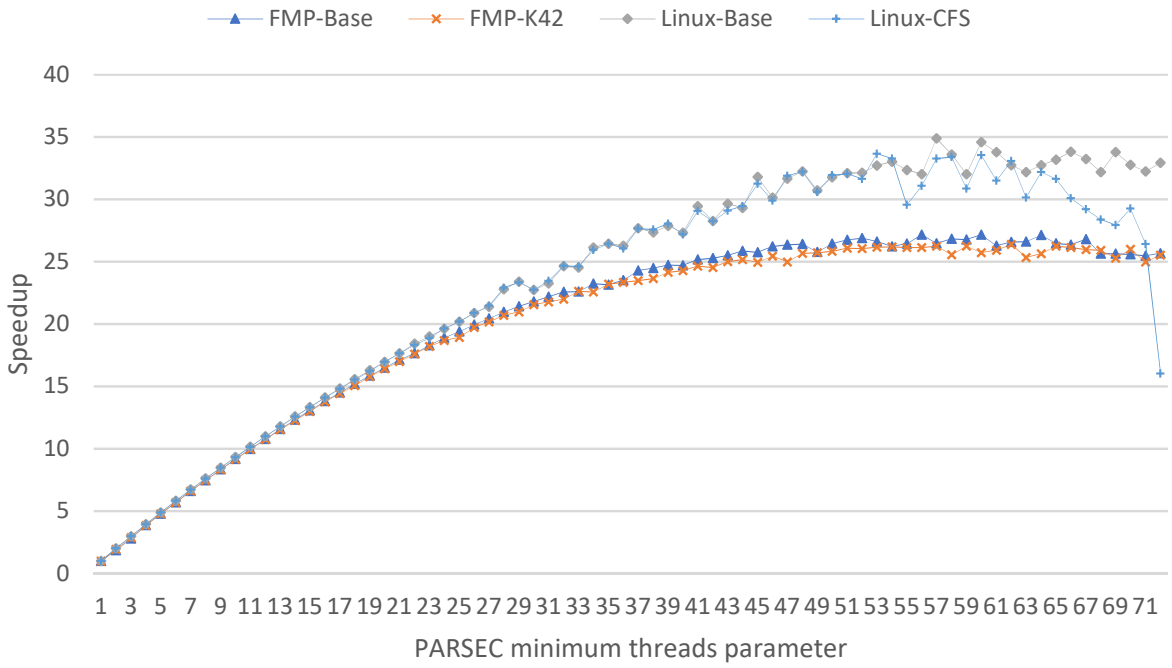

Fig. 3.11 Scalability of streamcluster with different runtime system settings

Fig. 3.12 shows the breakdown of execution time. The work in all settings scales well but the time for sync differs a lot. Previous study has shown that the inefficient barrier implementation in streamcluster can be a bottleneck [105]. Streamcluster uses a spin-then-block strategy for barrier by default. However, streamcluster does not have load imbalance problem, which means spinning can be much better than blocking in most situations. The barrier in streamcluster is implemented using mutexes and conditional variables provided by the POSIX thread library. In fact, the mutex implementation in Linux also uses a spin-then-block strategy. That is to say, although FMP and Linux use the same source code for barriers in streamcluster, the barriers in Linux can actually spin longer than FMP. Consequently, Linux can show better scalability than FMP. The collapse of scalability in Linux-CFS is caused by the load balancing. Since streamcluster is not imbalanced, load balancing cannot increase its performance at all. On the contrary, expensive thread migrations can harm the performance of synchronization.

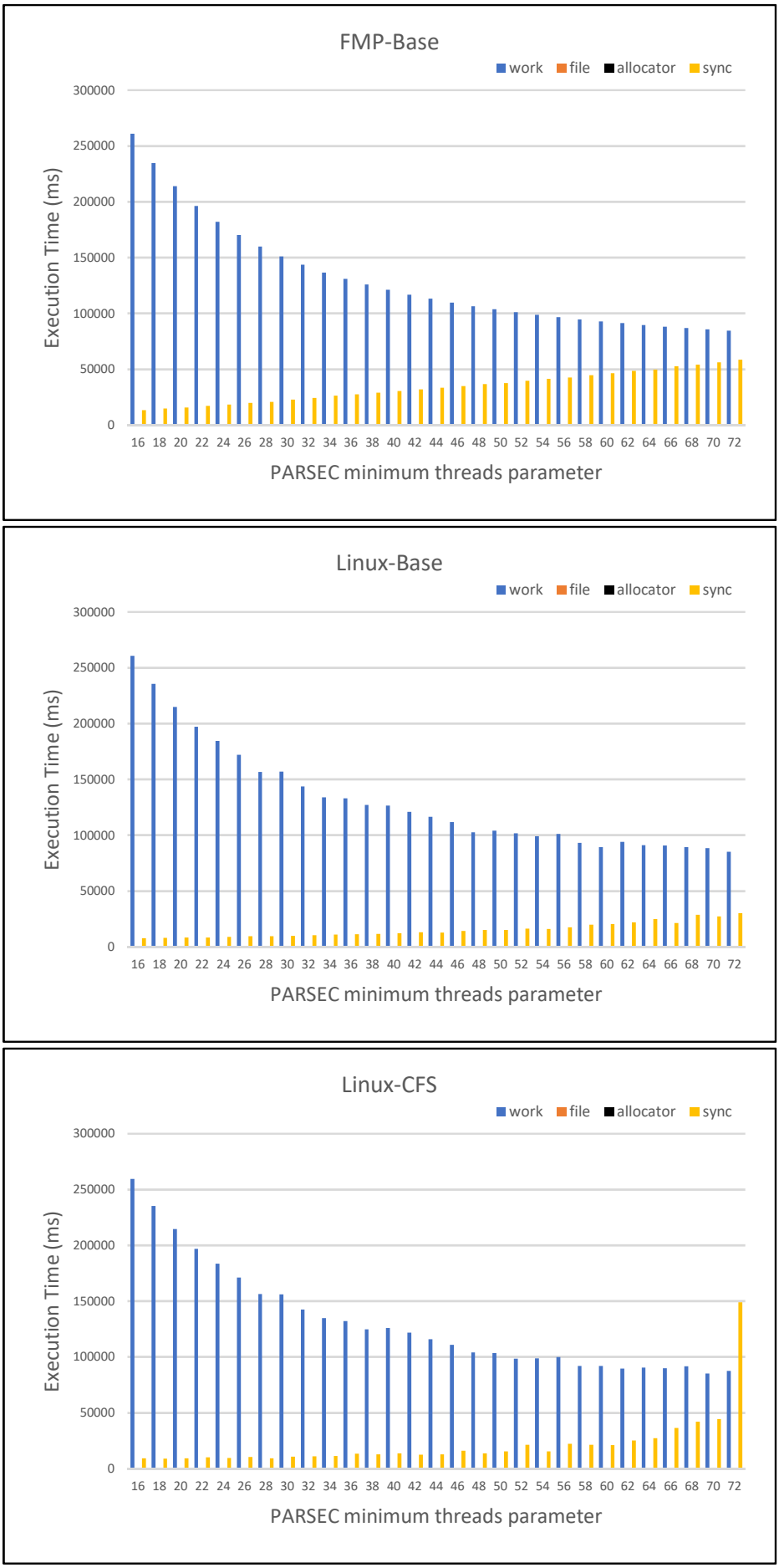

Fig. 3.12 Breakdown of streamcluster execution time

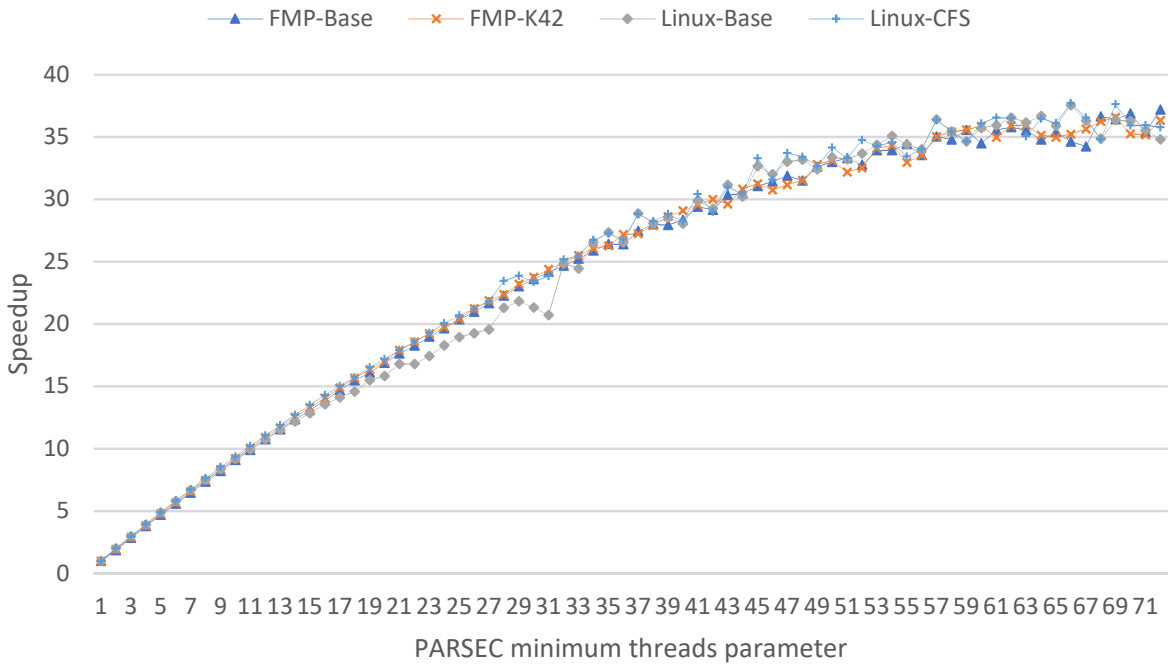

Fig. 3.13 Scalability of streamcluster with spin barriers

To rule out the difference caused by the inefficient barriers, we have measured the scalability again with spin barriers. As shown in Fig. 3.13, the scalability is greatly improved with spin barriers and the trends for all the settings have become very similar.

3.4.5 Dedup

Dedup is an application for multithreaded data compression. It is communication intensive and uses the pipeline model. More than 150,000 locks and thousands of conditional variables are used for synchronization. Unlike those fork-join applications, the number of threads running simultaneously in dedup can be as 3 times as the thread parameter.

We have found that Linux-Base setting cannot be used for dedup due to a CPU starvation caused by spinlock. We believe that it is because Hoard for Linux is not designed to work under fixed-priority preemptive scheduling. For fork-join applications, it is fine since there is only one worker thread for each core. For pipeline application like dedup, if a thread spins to wait a lock held by another thread on the same core, the starvation can happen. Therefore, we use another setting called Linux-Base-pt for dedup. In Linux-Base-pt, the Hoard memory allocator in Linux-Base is replaced by the ptmalloc from glibc.

Fig. 3.14 shows the scalability of dedup. Although dedup uses many spinlocks for synchronization, FMP-K42 has worse performance than FMP-Base. It is because that most spinlocks in dedup are used to protect elements in a hash table. Typically, an element in

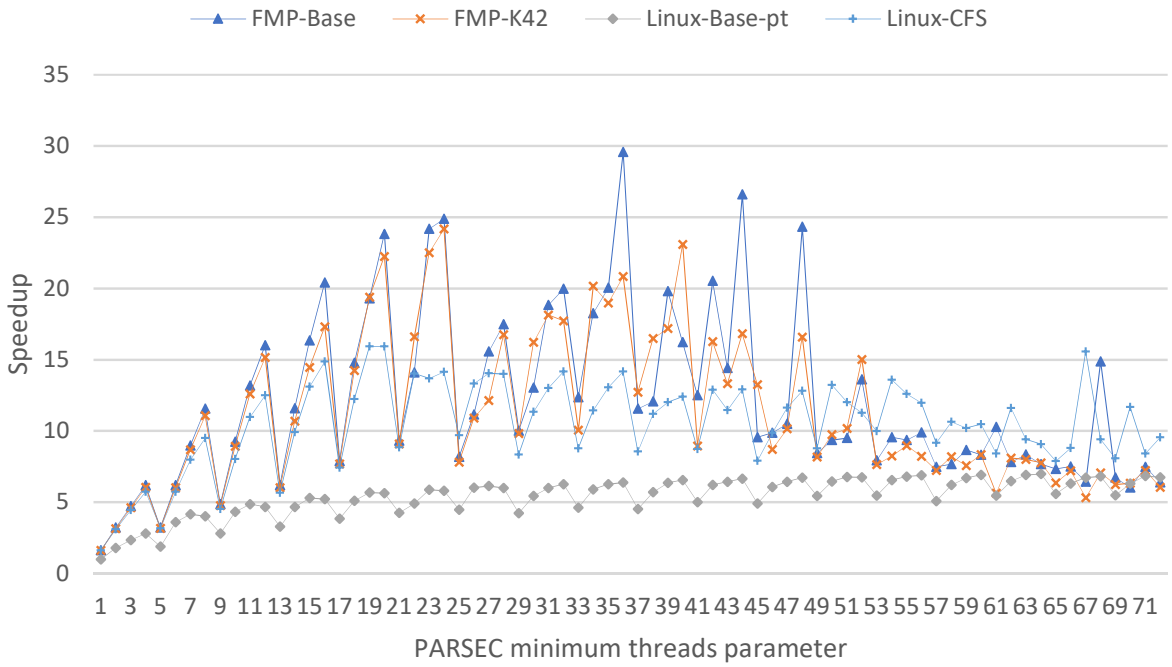

Fig. 3.14 Scalability of dedup with different runtime system settings

that hash table will not be accessed by multiple threads at the same time. As described in Section 3.3.1.5, K42 actually has the worst throughput if there are less than 3 threads acquiring the same lock. The Linux-Base-pt shows the worst scalability and it is suggested that the ptmalloc memory allocator has become a bottleneck. The scalability of FMP-Base is about twice as good as Linux-CFS when the thread parameter is smaller than 50. Linux-CFS has the best performance at last.

The analysis of scalability for pipeline application is very complex because it is not determined by a single path like fork-join applications [91]. Though, we can still shed some light by breaking down the execution time. We have summed up the average execution time of every pipeline stages and the result is shown in Fig. 3.15. The time in file and allocator is much short in FMP-Base, which should be the main reason why FMP has better scalability in most cases. The load balancing mechanism in Linux-CFS keeps the sync time to a relatively small value, and we believe that is why Linux-CFS can provide better performance at last. However, we can also see higher work time in Linux-CFS and it is suggested that thread migrations can slow down the execution.

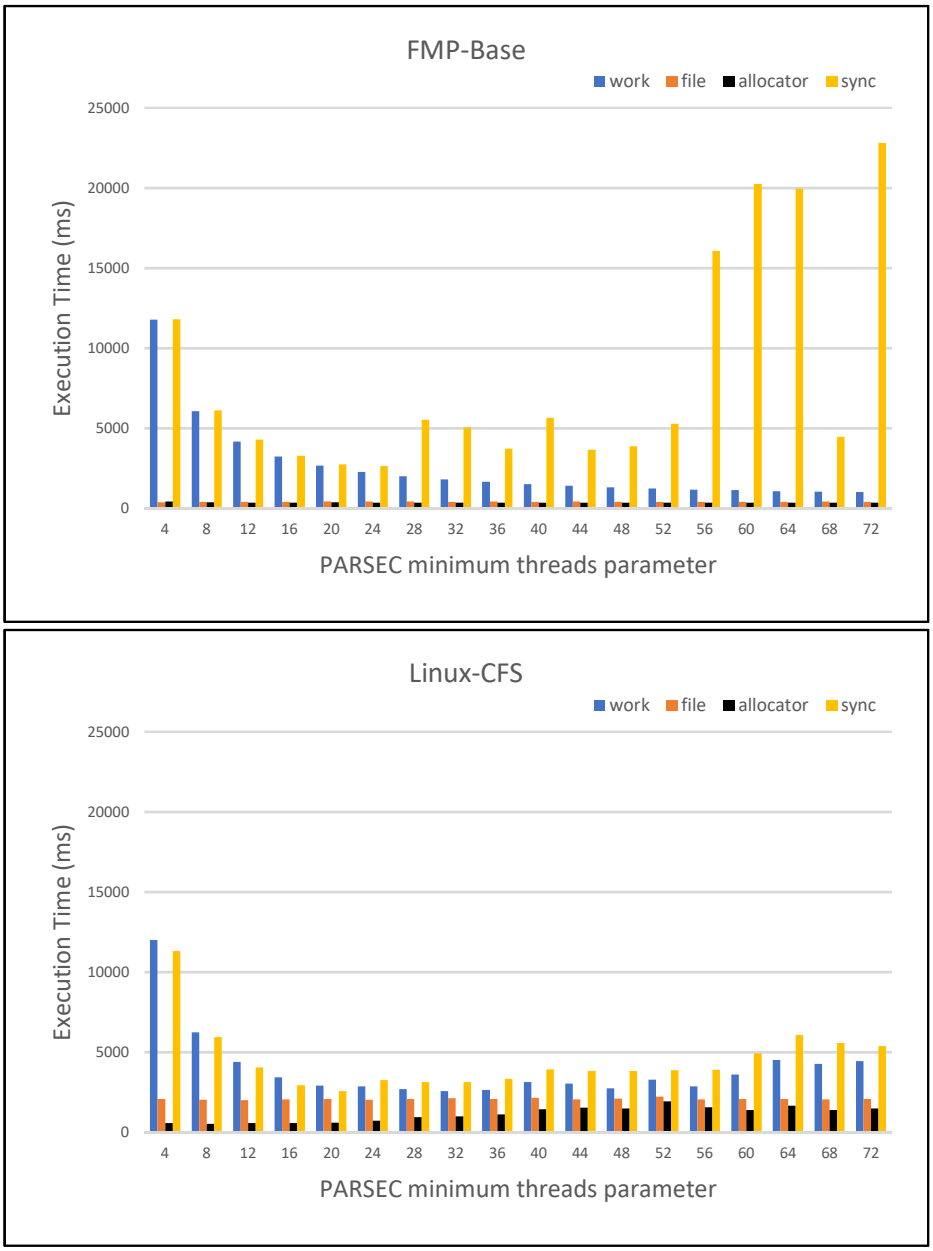

Fig. 3.15 Breakdown of dedup execution time

3.5 Conclusions

We have presented an experiment environment based on the TOPPERS/FMP kernel and the 72-core TILE-Gx72 processor. It is the first, to our knowledge, publicly released open source testbed for evaluating traditional multi-core RTOS on an off-the-shelf embedded many-core processor. By a comparative analysis of RTOS-based and Linux-based runtime systems,

we have identified several bottlenecks in traditional RTOS, such as non-scalable spinlock and memory allocator implementations. These bottlenecks have already been addressed in Linux and the methods to avoid them in RTOS are proposed in this chapter. Finally, performance evaluation on RTOS and Linux is performed using the PARSEC benchmark suite. The results show that, although traditional RTOS is not designed for executing high performance applications, it can deliver better scalability than Linux in many cases, with proper optimization. Therefore, since previous study has shown that Linux can scale well on many-core processors with tens of cores, we believe that traditional RTOS like TOPPERS/FMP can also be a good choice for embedded many-core processors.

Chapter 4

ESPROF: Generic Profiling Infrastructure for Embedded Multi/Many-Core

4.1 Introduction

The research of FMP-MC in Chapter 3 has shown that a traditional multi-core RTOS-based platform, although not designed for running high-performance applications, can be optimized to deliver good scalability for parallel computing on a many-core processor. Nevertheless, the actual scalability can still be limited by bottlenecks inside the application. Performance analysis tools are essential for identifying those bottlenecks. Most existing tools for embedded systems are focused on real-time applications [119, 7, 41, 9, 3, 54, 41, 130, 96]. Many advanced tools for analyzing parallel applications have been created in the field of high-performance computing (HPC) [1, 51, 114, 66, 67]. However, these tools cannot be used in embedded systems due to differences in characteristics between HPC and embedded systems. Consequently, the lack of performance analysis tools has become one of the major obstacles when developing high-performance applications for embedded systems.

In this chapter, *esprof*, a generic infrastructure for developing profiling tools is presented. It allows users to effortlessly create customized profilers that can meet the requirements of measuring high-performance embedded applications. Unlike configurable profilers using predefined settings, profilers on *esprof* can be flexibly implemented in Python. *ecg*, a scalable call graph profiler has been developed as an example. It is based on an optimized algorithm which generates different measurement code according to the code structure information provided by our infrastructure. *ecg* can output profiling results in GPTL [104] data format

which is supported by a popular parallel application analysis tool called ParaProf [18]. We evaluate *ecg* and GPTL's call graph profiler by measuring an application from the PARSEC benchmark suite [20] on a 36-core platform. *ecg* has successfully identified bottlenecks in the application while GPTL cannot. The results also show that *ecg* can provide much higher accuracy with very low overhead.

The rest of this chapter is organized as follows. In Section 4.2, the characteristics and challenges of existing performance analysis tools used in HPC and embedded systems are discussed. Based on the discussion, we analyze the requirements of profiling tools for high-performance embedded applications in Section 4.3. *esprof*, the proposed infrastructure to meet these requirements is explained in Section 4.4. We then describe the design and implementation of *ecg*, a scalable and optimized call graph profiler on *esprof* in Section 4.5. The *ecg* profiler and GPTL's call graph profiler are evaluated on a many-core platform in Section 4.6. Finally, this chapter is concluded in Section 4.7.

4.2 Review of Existing Tools

Many profiling tools to gather performance metrics of application for further analysis have been created for HPC and embedded systems. We will review the characteristics and methodologies of these tools in this section. The challenges of using them for high-performance applications in embedded systems are also discussed.

4.2.1 Target Embedded Systems

In this research, we focus on embedded systems for applications with both high-performance and real-time requirements.

Some embedded many-core processors designed for these applications have been released [99]. These processors are usually resource-constrained (e.g. the 64-core Epiphany-IV processor have 2MB local memory in total [93]). Several of them support external DDR memory but the cost of access can be very expensive. The 256-core Kalray MPPA processor, for instance, must always use DMA engines to access the external memory [97]. Although different hardware architectures and instruction sets are used, most, if not all, of them have compiler for standard C language. C++ is also (partially) supported on some target systems.

Traditional general-purpose platforms (e.g. Linux-based HPC environment) are not designed for achieving high scalability and predictable timing behavior at the same time. Researchers are proposing new software platforms (e.g. operating systems and parallel

programming frameworks) in order to meet these requirements [99]. New profiling tools are required to analyze performance on those platforms.

4.2.2 Data Collection Methods

There are mainly two ways to collect performance data: sampling and instrumentation. Each method has its own advantages and disadvantages.

A sampling-based profiler measures application by taking statistical samples. It typically uses a cyclic handler to interrupt the execution of application and record performance information. The application remains unmodified and thus this method has very little impact on its performance. Since the profiler stands outside the application, it can be, at least in theory, used as a general tool to measure any kind of program (e.g. user-mode application, kernel driver or even interrupt handler). The overhead of profiling basically depends on the sample rate rather than the measured application. The main limitation of sampling is that the captured profiles are flat and approximate. Although it can show results such as the hot spots in program and the occurrence rate hardware events, it is not suitable for detailed analysis. Further, in order to improve accuracy, the user has to choose a proper sample rate. The data could be unreliable if the frequency is too low, while fast frequency will introduce high overhead.

An instrumentation-based profiler, in contrast, gathers performance metrics by inserting measurement code into the application. It allows user to capture interested performance information in detail. For example, profiles like a high-accurate call graph and the exact number a function has been invoked, can only be acquired by instrumentation. The overhead of profiling is determined by the execution time of inserted code, which can be huge compared to sampling, especially when measuring an application visiting instrumentation points frequently. Therefore, it is essential to optimize the implementation of measurement code as possible.

Application can be instrumented either dynamically at run time or statically at compile time [92]. Dynamic instrumentation has advantages such as avoiding recompilation and allowing user to measure program generated during execution, but requires the application executed from RAM. Static instrumentation can be done at the level of source code or binary (machine) code. Source code instrumentation is target-independent but language-specific, while binary instrumentation is language-independent but target-specific. Embedded systems barely generate code at run time and many of them are executed from ROM. There are a lot of different targets (architectures and operating systems) but almost all of them use C/C++ as programming languages. Hence, we believe the most suitable instrumentation method for embedded systems is to modify the source code statically.

4.2.3 Profiling Tools for HPC

Performance analysis tools for HPC systems have been actively researched for several decades. HPCToolkit [1], Periscope [51], Scalasca [50], TAU [114] and Vampir [66] are some well-known examples. Although these tools are designed and well-optimized for multi/many-core applications, they cannot be used in embedded systems because HPC and embedded systems differ greatly in their characteristics.

HPC applications are usually developed with cross-platform programming interfaces such as OpenMP [37] and MPI [46] for portability and maintainability. The runtime overhead of these HPC-oriented interfaces can be very expensive for the computing power of embedded processors. Therefore, parallel applications in embedded systems tend to use native tasks provided by RTOS or various lightweight thread libraries [28].

The scale of code and data for HPC is so large that recompiling and re-executing can be extremely time-consuming. In order to avoid recompiling, many tools put all support measurement functionalities into the application during instrumentation, and let user to choose at run time (for example, through a setting file) [67]. However, this approach also significantly increases the complexity, footprint and implementation cost of profiling tools. Further, user may also want to collect more data as possible by enabling more instrumentation points for each execution of long-running applications, which can lead to high measurement overhead. On the other hand, the scale of embedded applications is relatively small due to limited resources available (e.g. memory, battery) and the deadlines are very short compared to the execution time of HPC applications. It would be better to provide multiple small profiling tools for embedded systems, each for measuring one aspect of the application with small footprint and overhead. Developers can then repeatedly measure the application with different tools to get interested information.

Performance profiles for HPC applications are often stored in a scalable file system or database due to the huge amount [67]. Most embedded systems only have low-speed small file systems, or no file system at all. Therefore, collected data should be stored in memory whenever possible. By using a small profiling tool, the usage of memory can be reduced.

Many advanced tools have been developed for analyzing and visualizing the performance data of HPC applications. For example, ParaProf [18] is a famous cross-platform profile analysis tool supporting many different data formats. It is extensible and scalable and can be used for large-scale parallel analysis such as a 512-processor application. Embedded application developers can also use this kind of powerful tools for analysis, if the profiler is able to generate data in compatible format.

4.2.4 Profiling Tools for Embedded

Sampling-based profilers have been supported by many RTOSes because of the versatility [119, 7, 41]. Some development tools also provide low-level profilers using hardware trace technology of the target processor [9, 3, 54]. Functionalities of these hardware profilers are very similar to the sampling-based software profilers but the overhead is much smaller.

As mentioned above, instrumentation can provide details and accuracy that sampling cannot, which is very useful for fine-tuning and optimization. Current RTOSes mainly use this method for event tracing rather than profiling performance metrics [41, 130, 96]. High-performance applications, unlike traditional real-time applications, can produce numerous event logs during execution (e.g. over 100,000 entries per core). It is very difficult, if not impossible, to analyze performance characteristics from this amount of logs because a lot of noises are included. Besides, most target systems only have small memory size for storing logs. Summary-based profilers are needed to detect hot spots and filter unimportant information before detailed analysis.

4.2.5 Source Code Instrumentation Tools

Some compilers can generate instrumentation code for profiling. For example, GCC has `-finstrument-functions` option to add calls at entry and exit of functions. Since it is a compiler-dependent functionality, profilers based on this approach have poor portability among compilers from different vendors. Moreover, the developer has very limited control over the instrumentation process. In the case of GCC, all instrumented functions share the same (and fixed) hook functions (i.e. `__cyg_profile_func_enter/exit`). GPTL (General Purpose Timing Library) [104] includes a multi-threaded call graph profiler using this GCC option. Its data format is officially supported by ParaProf. Although GPTL is designed for UNIX-like OSes, its OS-dependent interface is very simple and portable. We have extended the interface for RTOS so it can be used to compare with our proposed profiling tool.

While profilers using source code instrumentation are presented in some papers [32, 137], most of those tools have not been released as of writing. TAU [114], to our knowledge, is the only publicly available profiling tool using this technique for C/C++.

In [49], how to adapt TAU instrumentor for use as a generic and configurable tool is described. However, the proposed approach uses predetermined rules and keyword (string) substitution for code generation. The lack of flexibility makes it unsuitable for implementing profilers using algorithms to generate different code at each instrumentation point (e.g. `ecg` profiler in Section 4.5).

An alternative technique to insert measurement code is using AOPL (Aspect-Oriented Programming Language) such as AspectC [35] and AspectC++ [118]. However, AspectC is not publicly available and AspectC++ cannot be compiled with pure C compilers as of writing. Further, AOPL typically requires a runtime library, which can introduce additional overhead and thus make the measurement results less accurate.

Due to the above limitations, user often has to manually insert code per application for performance analysis, which is time-consuming and hard to maintain. A generic and flexible instrumentor can be very helpful for creating new profiling tools.

4.3 Requirements Analysis

We will now analyze the requirements of profiling tools for high-performance embedded applications, based on the discussion in previous section.

Small tool is preferred. The scale of embedded applications is relatively small, and thus it is unnecessary to use complex monolithic tools to avoid recompiling and re-executing like HPC systems. Small profiling tool that measures a specific aspect of the application is easy to implement and can provide excellent reusability. It also has smaller footprint and overhead, which is suitable for resource-constrained embedded systems.

Static source-level instrumentation. Profiler using instrumentation can provide high-accurate details for fine-tuning and optimization. Static instrumentation should be used because many embedded applications are executed from ROM where code cannot be modified at run time. C/C++ have become de facto standard programming languages for embedded systems while the target platforms are highly diversified. Therefore, target-independent source-level instrumentation is more generally applicable.

Scalable in-memory data structures. File systems in embedded systems are usually slow and simple. Many target platforms have no file system at all. Unlike in HPC systems, profilers cannot rely on scalable file systems to store the performance data efficiently. All collected data should be stored in memory instead. Data structures shared by multiple cores will cause bottlenecks easily, especially on many-core platforms [23]. Developer should use thread-local storage or lock-free algorithms whenever possible to improve scalability. After measuring an application, the in-memory data must be outputted to the host (e.g. via serial communication). If a profiler uses performance metrics similar to some HPC-oriented tool, it would be better to output data in the same format, which allows user to make use of existing advanced analysis tools. Sometimes it may be difficult to directly construct the data format (e.g. a database file) from the target side. The profiler can output in-memory data as simple format like plain text at first, and then use a converter on the host side to get desired format.

Portability for different targets. Targets of embedded systems can differ in many ways, including but not limited to processor architectures, RTOS APIs and thread libraries for parallel programming. Profilers should be implemented with portability except for those created to measure target-dependent metrics. This usually requires developer to define a target abstraction layer and provide implementation for each supported target.

A common infrastructure. Developing multiple small profiling tools from scratch will cost a lot of effort. A well-designed generic infrastructure can greatly increase productivity and reduce development costs. Examples of similar functionalities in different profiling tools, which should be handled by the infrastructure, include source parsing, code instrumentation, building system and tool integration. Further, the same application is likely to be measured multiple times with different tools in order to get all interested information. The common infrastructure should allow user to easily switch between tools.

4.4 esprof: the Profiling Infrastructure

We present *esprof*, a generic infrastructure that allows users to flexibly and effortlessly develop profiling tools that meet the requirements described in previous section. Main characteristics of *esprof* are listed as follows.

Minimal footprint. Small tools with low overhead are preferred for resource-constrained embedded systems. Unlike HPC-oriented infrastructure (e.g. Score-P [67]) using a runtime designed for the standard environment, *esprof* itself does not include a common runtime. User (i.e. tool developer) has full control over the inserted measurement code. Therefore, *esprof*, in theory, can be used on any platform providing a C compiler, without the additional footprint of implicit code.

Flexible instrumentation. Developer can control the instrumentation process of *esprof* using full-featured programming languages (e.g. Python). The improved flexibility allows more tools (especially those generating code with advanced algorithms) to be supported compared to existing rule-based generic instrumentors [49]. It is also possible to take advantage of the *instrumentor knowledge* for further optimization.

Easy integration. A disadvantage of using multiple tools is recompilation and integration for each tool. *esprof* includes a unified mechanism to manage and integrate profiler projects. User can easily switch between different tools (typically by a single shell `source` command). Therefore, it is effortless to write a script to do compilation for all desirable tools.

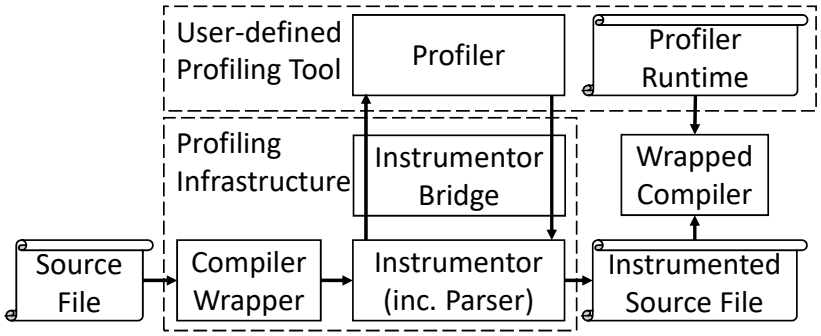

Fig. 4.1 Architecture of *esprof*

4.4.1 Architecture

The architecture of *esprof* is shown in Fig. 4.1. We will now explain its components.

4.4.1.1 Compiler Wrapper

A compiler wrapper command, called *esprof.py*, is provided as a facade for instrumenting source files. It works as a prefix to the compile commands (e.g. `esprof.py g++ -c foo.c`).

The wrapper invokes related tools like instrumentor and user-defined profiler to modify the input source code. It will then execute the wrapped compiler, with proper parameters required by the profiler, to compile the instrumented source file. The wrapper's behavior can be configured through environment variables, as described later in this chapter.

4.4.1.2 Instrumentor

Instrumentor in *esprof* is a general-purpose tool for inserting code snippets at desired locations in a source file. Unlike special-purpose instrumentors which only generate measurement code for particular profilers, our instrumentor does not include any concrete code generation rule internally. Instead, it will send the information of code to, and receive instrumentation requests from the user-defined profiler.

Instrumentor typically uses a parser to analyze the source code statically. The interface of parser can be based on line number or AST (Abstract Syntax Tree). Line-based parser is easy-to-use due to its simplicity but also has limitations. For example, it cannot insert code between executions of `a()` and `b()` for `f=a()+b();` because they are in the same statement. In some cases, user may work around the limitations by using code formatter or rewriting the code manually. AST-based parser is complicated but can provide more details and manipulate the code more precisely. It allows the instrumentor to transform `f=a()+b();`

to `Ra=a();Rb=b();f=Ra+Rb;` for further processing. PDT (Program Database Toolkit) [78] is a famous tool for line-based parsing, which uses a commercial software called EDG C++ Frontend [40]. ROSE [102] is an open source compiler infrastructure for building AST-based tools such as code analyzers.

We have developed *esprof-inst*, a prototype of general-purpose instrumentor for *esprof*. *esprof-inst* reuses some code from TAU instrumentor (based on PDT) for parsing. It implements the protocol of instrumentor bridge (described later in this section) to support flexible instrumentation.

4.4.1.3 Instrumentor Bridge

Instrumentor bridge is a component that defines and implements the protocol for communication between instrumentor and user-defined profiler.

The bridge requires instrumentor to generate a JSON file called instrumentor knowledge. This file contains information such as the name of source file, the prototype of functions, and the ID of each instrumentation point. Currently supported instrumentation points include the header of source code, entry and exit points of functions, and before and after callees in a function. A sample source file and its corresponding instrumentor knowledge are shown in Fig. 4.2 and Fig. 4.3. Functions and callees are sorted by their locations in the source code. ID of instrumentation points can be unsorted and noncontiguous.

After the instrumentor knowledge is generated, it is passed to the user-defined profiler by the bridge. The profiler should then generate a list of instrumentation requests in JSON. Each request includes ID of an instrumentation point and a code snippet. For example, the JSON string `{'requests':{0:'#include "profiler.h"'}}` can be used to add a header file `profiler.h` for source file in Fig. 4.2. The bridge will send generated requests back to the instrumentor in order to modify the code accordingly.

```
// Instrumentation Point 0

extern int f1(int k);

void foo() {
// Instrumentation Point 1
  int a = 1;
// Instrumentation Point 2
  a = f1(a);
// Instrumentation Point 3
  if (a > 0) {
// Instrumentation Point 4
      f1(5);
// Instrumentation Point 5
// Instrumentation Point 6
      return;
  }
// Instrumentation Point 7
}

void bar() {
// Instrumentation Point 8
// Instrumentation Point 9
}
```

Fig. 4.2 A sample source file named foo.cpp

```
{
'filename': '/path/to/foo.cpp',
'header': 0,
'functions': [
 {'prototype': 'void foo()',
  'entry': 1, 'exits': [6,7],
  'callees': [
   {'prototype': 'int f1(int)',
    'before': 2, 'after':3},
   {'prototype': 'int f1(int)',
    'before': 4, 'after':5}]},
 {'prototype': 'void bar()',
  'entry': 8, 'exits': [9]}]
}
```

Fig. 4.3 Instrumentor knowledge for foo.cpp

```
from instrumentor import instrumentor

instrumentor.header_snippet = f'/* {instrumentor.filename} header */'

for func in instrumentor.functions:
    func.entry_snippet = f'/* {func.name} enter */'
    func.exits_snippet = f'/* {func.name} exit */'
    for callee in func.callees:
        callee.before_snippet = f'/* before call {callee.name} */'
        callee.after_snippet = f'/* after call {callee.name} */'

instrumentor.instrument()
```

Fig. 4.4 Example of a dummy profiler script in Python

4.4.1.4 User-defined Profiler

The JSON-based protocol of instrumentor bridge is portable and extensible. It allows profilers and instrumentors to be implemented in almost any programming language. New types of source code information and instrumentation points can be easily added as necessary.

For now, a helper library for developing profiler in Python is included in the bridge. It provides an easy-to-use interface by hiding the details of handling JSON data. A dummy profiler script in Python that inserts comment at each instrumentation point is shown in Fig. 4.4 as an example.

4.4.2 Structure of a Profiler Project

A workspace folder under the folder of *esprof.py* is used to hold multiple profiler projects. Each user-defined profiler project corresponds to a dedicated folder under the workspace folder, which usually consists of following files.

- **Environment files for supported targets.** Several environment variables must be set to use profiler built on top of *esprof*. `ESPROF_BIN` is the path of *esprof.py*. `PROFILER_BIN` is the path of user-defined profiler (by default, Python script `profiler.py` under the project folder). `PROFILER_SRC` is the folders including source files of profiler (by default, `src` under the project folder). `PROFILER_OPT` is the extra compiler options for instrumented source files. `PROFILER_OBJ` is the object files that should be additionally compiled and linked. Since variables like `PROFILER_OPT` and `PROFILER_OBJ` are usually target-dependent, profiler developer should provide environment file for each supported target.

- **User-defined profiler.** It is an executable file (usually, a Python script) that uses the instrumentor bridge to insert necessary measurement code snippets.
- **Profiler runtime.** It usually includes the data structures, macros and functions used by the instrumented code. When developing a portable profiler, header and source files of its runtime can be divided into target-independent part and target-dependent part. The target-independent part should declare an interface (i.e. abstraction layer) to be implemented in each target-dependent part.

4.4.3 Integrating into Existing Application

Our infrastructure can be easily integrated into an existing application by adding environment variables `ESPROF_BIN` (as prefix to compile command), `PROFILER_SRC` (as source folders) and `PROFILER_OBJ` (as object files) to proper locations in the Makefile of application to be measured. The application developer can use a profiler by simply importing its environment file for corresponding target before building. This approach also allows user to easily switch between different profilers based on *esprof*. If no environment file for profiler is imported, these variables are empty and the application will just be built as usual.

4.5 ecg: a Call Graph Profiler on esprof

As an example, the development of *ecg*, a portable and scalable call graph profiler on *esprof*, is described in this section. Its algorithm is optimized by making use of the instrumentor knowledge provided by *esprof*. A limitation of this optimization is that calling with dynamic function pointers cannot be measured, because the static information of callees are used to generate measurement code.

4.5.1 Data Structures and Algorithm

A sequence of function calls (during the execution of one thread) is called a calling context. Performance profiles of each calling context can be stored in a CCT (Calling Context Tree) [5]. A CCT can distinguish the same function called in different contexts. Therefore, it provides more accurate and useful information compared to context-insensitive call graphs.

Data structures of our CCT variant are shown in Fig. 4.5. Each node in a CCT (i.e. a calling context) is represented by a *CallRecord*, which includes following fields.

- **address** is the starting address of measured function.

```
struct CallRecord {
  void       *address;
  Metrics     metrics;
  CallRecord *children;
  int         nchildren;
};

struct TLS {
  CallRecord *current;
  CallRecord *parent;
};
```

Fig. 4.5 CCT data structures

- **metrics** is used to store performance data. For example, it may include the number of times a function has been called and total execution time spent in that function.
- **children** and **nchildren** are the base address and size of a *CallRecord* array. Each element in that array corresponds to a call site (the location where a function is called) in the measured function.

A thread-local variable of *TLS* is used to keep track of calling context for each thread. It consists of pointers to the current CCT node (i.e. callee) and its parent node (i.e. caller).

Our CCT can be constructed at run time by inserting necessary code into measured application at following points.

- **Start and exit of a thread.** CCT and *TLS* are initialized when a thread starts. The root node of CCT includes a single child node for the start routine. Metrics of this child node are recorded by taking snapshots at start and exit points. The corresponding pseudocode is shown in Algorithm 1.
- **Entry and exit of a function.** *TLS* is checked at the entry of a function. All instrumented code in the function will be skipped for unmatched current node or recursive calls. Unmatched current node, which could be caused by calling from uninstrumented function, means that the *CallRecord* is not for current function. Metrics of recursive calls are recorded as a whole in the first level of recursion, in order to reduce overhead and memory usage. In normal case, the parent node in *TLS* is switched to current node for further execution. The array for callees (children nodes) will also be allocated and initialized when necessary. The corresponding pseudocode is shown in Algorithm 2.

Algorithm 1 Start and Exit of a Thread

```
1: procedure THREADSTART(startAddress)
2:     Allocate two CallRecord c0, c1
3:     c0.address ← NULL                                   ▷ Root node
4:     c0.children ← &c1
5:     c0.nchildren ← 1
6:     c1.address ← startAddress
7:     tls.parent ← &c0
8:     tls.current ← &c1
9:     m0 ← snapshot of metrics
10:    Push c1 and m0 onto stack
11: end procedure
12: procedure THREADEXIT
13:    Pop c1 and m0 from stack
14:    m1 ← snapshot of metrics
15:    Update c1.metrics using m0 and m1
16: end procedure
```

Algorithm 2 Entry and Exit of a Function

```
1: procedure FUNCTIONENTRY(startAddress, nChildren, initCode)
2:     if tls.current.address ≠ startAddress then        ▷ Current node does not match
3:         Skip all instrumented code in this function
4:     else if tls.parent.address = startAddress then    ▷ Recursive call
5:         tls.parent.metrics.nrecursive++               ▷ If metrics count the recursive calls
6:         Skip all instrumented code in this function
7:     else
8:         if tls.current.children = NULL then
9:             Allocate CallRecord array c[nChildren]
10:            Execute initCode to initialize c          ▷ E.g. set address of each child (callee)
11:            tls.current.children ← c
12:            tls.current.nchildren ← nChildren
13:        end if
14:        parent ← tls.parent
15:        tls.parent ← tls.current                      ▷ Use current node as new parent
16:        Push parent onto stack
17:    end if
18: end procedure
19: procedure FUNCTIONEXIT
20:    Pop parent from stack
21:    tls.parent ← parent
22: end procedure
```

Algorithm 3 Before and After a Callee

```
1:  procedure BEFORECALLEE(index)
2:      m0 ← snapshot of metrics
3:      cur ← &tls.parent.children[index]
4:      tls.current ← cur
5:      Push m0 and cur onto stack
6:  end procedure
7:  procedure AFTERCALLEE
8:      Pop m0 and cur from stack
9:      m1 ← snapshot of metrics
10:     Update cur.metrics using m0 and m1
11: end procedure
```

- **Before and after a callee.** For each callee in a function, the index in the children array is determined by the order of call sites. Before jumping to a callee, the current node in *TLS* is set to corresponding child node. Metrics are recorded by taking snapshots before and after that callee. The corresponding pseudocode is shown in Algorithm 3.

When execution of the instrumented application is finished, CCTs with performance profiles for each thread are generated.

The characteristics of our algorithm, compared to the original CCT paper [5], are explained as follows.

ecg is designed for target-independent source code instrumentation while [5] uses binary instrumentation for SPARC processors.

In [5], child nodes are individually initialized and allocated according to the information at run time. In *ecg*, the initialization code of child nodes for each instrumented function is statically generated and passed as argument of Algorithm 2, which has lower overhead.

In most cases, only metrics of certain regions in a system (e.g. a specific application) are interesting to the user. Instrumenting common libraries like middlewares and kernel services, however, will affect performance of the whole system. Sometimes the instrumentation is simply impossible due to lack of source code for these libraries. Unlike [5], *ecg* collects performance data at the caller side (before and after each call site as shown in Algorithm 3) to reduce overhead. The accuracy is also improved because even if a callee function is uninstrumented, its metrics can be measured and recorded in the CCT.

4.5.2 Implementing the Profiler

ecg, a profiler for the proposed CCT design, has been implemented on the *esprof* infrastructure. Metrics of *ecg* include the number of invocations, the number of recursive invocations

```
from instrumentor import instrumentor

for func in instrumentor.functions:
  if len(func.callees) > 0:
    init_code = ''
    for idx in range(len(func.callees)):
      init_code += f'__ecg_current->children[{idx}].address = ' + \
        f'(void*){func.callees[idx].name};'
      func.callees[idx].before_snippet = f'_ECG_CALLEE_BEFORE({idx});'
      func.callees[idx].after_snippet = '_ECG_CALLEE_AFTER();'
    func.entry_snippet = f'_ECG_FUNCTION_ENTER(' + \
      f'(void*){func.name}, {len(func.callees)}, {{{init_code}}});'
    func.exits_snippet = '_ECG_FUNCTION_EXIT();'

instrumentor.header_snippet = '#include <esprof-callgraph.h>'
instrumentor.instrument()
```

Fig. 4.6 Python script for *ecg* profiler

and wallclock timing statistics, which are compatible with the performance data format of GPTL [104].

esprof provides sufficient code structure information required by *ecg*'s algorithm as instrumentor knowledge. The instrumentor bridge allows user to flexibly access instrumentor knowledge and insert different code for each instrumentation point accordingly with a Python script.

For portability, the implementation of *ecg* is divided into target-independent part and target-dependent part.

4.5.2.1 Target-independent Part

ecg defines a target abstraction layer which requires following functions to be provided in each target-dependent part.

- `_ECG_TLS* _ecg_get_my_tls()`: This function returns the pointer to the *TLS* variable of the caller thread.
- `_ECG_NODE* _ecg_alloc_nodes(size_t num)`: This function is used to allocate specified number of CCT nodes in a row for *CallRecord* data.
- `unsigned long _ecg_get_time()`: This function returns the timer value of a monotonic clock. The resolution (e.g. in nanoseconds) is decided by the target-dependent side.

```
PROFILER_DIR=$(dirname $(readlink -f ${BASH_SOURCE[0]}))
source ${PROFILER_DIR}/../env.common
export PROFILER_OPT=-I${PROFILER_DIR}/include/fmp-tilegx
export PROFILER_OBJ=ecg-fmp-tilegx.o
```

Fig. 4.7 Environment file of *ecg* for FMP-MC

Instrumentation points in our algorithm can be trivially implemented using these functions and thus we omit the details. We implement them in macros rather than functions so the stack operations can be automatically handled by the compiler with smaller overhead.

Python script for the profiler, which generates and inserts measurement code for Algorithm 2 and Algorithm 3, is shown in Fig. 4.6. Since the thread library used by the application depends on the target, hooks for Algorithm 1 should be implemented in the target-dependent part accordingly.

4.5.2.2 Supporting a Many-core Target

We have provided the target-dependent part for FMP-MC. The target abstraction layer is implemented as follows.

- `_ECG_TLS* _ecg_get_my_tls()`: For each thread, FMP-MC uses a contiguous memory block to store its thread-local variables. A special register is used to hold the base address of that block. Therefore, the return value of this function can be simply calculated in a few cycles.
- `_ECG_NODE* _ecg_alloc_nodes(size_t num)`: For each core, a memory pool of CCT nodes is created. This function uses core index to allocate nodes from the pool of current core. It provides good scalability since no data structures are shared between cores.
- `unsigned long _ecg_get_time()`: This function just returns the value of hardware performance counter which only costs a single instruction to read.

The environment file for FMP-MC is shown in Fig. 4.7. `PROFILER_OPT` and `PROFILER_OBJ` are set for adding target-dependent include path and source file. `env.common` is included to set default values for other required environment variables. User can use the *ecg* profiler to instrument application for FMP-MC by simply importing this file before building.

We also use the syslog service in FMP-MC to output CCTs in GPTL format supported by ParaProf. A screenshot of using ParaProf to analyze the performance profiles is shown in Fig. 4.8.

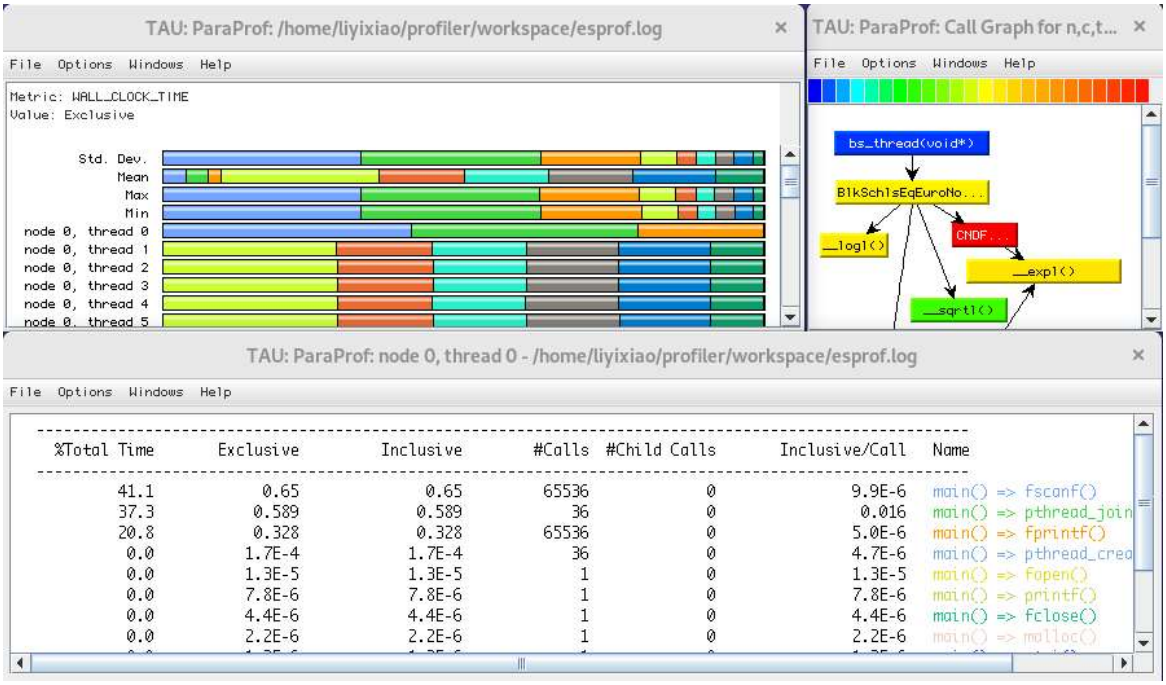

Fig. 4.8 A screenshot of ParaProf

4.6 Evaluation

In this section, we evaluate the *ecg* profiler and the call graph profiler in GPTL by measuring `blackscholes` application from the PARSEC benchmark suite with `simlarge` dataset. The target platform is FMP-MC on a 36-core many-core processor (TILE-Gx36).

GPTL uses GCC for instrumentation by providing the hook functions as decribed in Section 4.2.5. Two arguments, the address of current function and the address of call site, are passed to the hook. Other necessary information (e.g. the address of current and parent nodes) must be dynamically calculated and managed at run time using these arguments, which can lead to significant overhead. In contrast, the algorithm of *ecg* generates code for each point, as static as possible, using the instrumentor knowledge provided by *esprof*, in order to reduce this kind of dynamic run time overhead. Therefore, we can expect that *ecg* has better results than GPTL in most cases.

Performance profiles collected by these tools are shown in Table 4.1 and Table 4.2. Unit of total time is in milliseconds (ms). Unimportant entries (e.g. functions with total time less than 1ms) are omitted for clarity. Thread 0 is the main thread and thread 1 to thread 36 are worker threads. Profiles of worker threads are almost the same and thus we only show results for thread 1.

Table 4.1 Profiles collected using GPTL

Function Name	Total Calls	Recursive Calls	Total Time
=Thread 0=			
main	2	1	2221
=Thread 1=			
bs_thread	1	0	1236
BlkSchlsEqEuroNoDiv	546000	364000	987

Table 4.2 Profiles collected using *ecg*

Function Name	Total Calls	Recursive Calls	Total Time
=Thread 0=			
main	1	0	1580
fscanf	65536	0	650
pthread_join	36	0	589
fprintf	65536	0	328
=Thread 1=			
bs_thread	1	0	583
BlkSchlsEqEuroNoDiv	182000	0	580
__sqrtl	182000	0	36
__logl	182000	0	62.8
CNDF	182000	0	178
__expl	182000	0	61.6
CNDF	182000	0	178
__expl	182000	0	61.6
__expl	182000	0	64.1

GPTL can only measure instrumented functions (i.e. those defined inside the application) since GCC only supports hook for callees. *ecg*, as described in Section 4.5.1, measures from the caller side, and thus it can also collect metrics of uninstrumented library functions (e.g. `pthread_join` and `fscanf`).

According to the thread 0 profiles of *ecg*, over 60% of time is consumed in file I/O operations. This is a well-known bottleneck in `blackscholes` [79]. For thread 1, about 50% of time is consumed in the mathematics library. This is yet another bottleneck as described in Section 3.3.2.4 . Meanwhile, GPTL shows that `main` and `BlkSchlsEqEuroNoDiv` functions are called recursively. The results are misleading since there is no recursive function in the

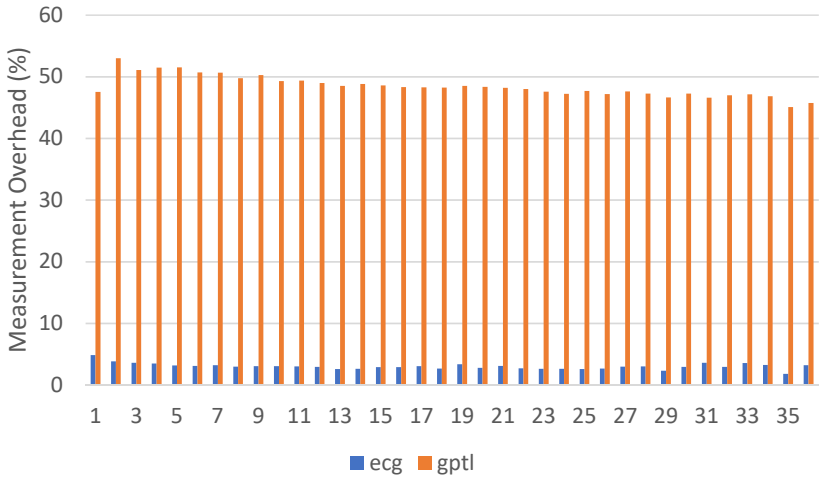

Fig. 4.9 Measurement overhead comparison

source code at all. We believe that this issue is caused by the instrumentation method used by GPTL. The GCC compiler will add code to call hooks of GPTL at entry and exit points of a function. However, if functions are optimized to partially share the entry code, they may be considered as the same function by GPTL.

We also compare the measurement overhead as shown in Fig. 4.9. The results indicate that both *ecg* and GPTL are scalable because the overhead does not increase as more cores are used. However, the overhead of *ecg* (4% in average) is negligibly small compared to GPTL (over 45%). By using static information to generate optimized measurement code, *ecg* can effectively reduce the dynamic run time overhead in constructing the call graph.

4.7 Conclusions

The lack of performance analysis tools has become one major obstacle in the development of high-performance applications for embedded systems. The requirements for suitable tools have been analyzed. *esprof*, a generic infrastructure for developing these tools is presented. It has a flexible and extensible architecture and provides most of the commonly required functionalities, such as code parsing and instrumentation, which allows user to create customized profiling tools effortlessly. As an example, *ecg*, a portable and scalable call graph profiler is implemented and optimized using *esprof*. We evaluated *ecg* and the call graph profiler of GPTL by measuring benchmark application on a many-core platform. The results show that *ecg* can provide much higher accuracy with very low overhead. Source code for our prototype implementation of *esprof* and *ecg* can be found in [75].

Chapter 5

Conclusions and Future Trends

5.1 Summary of Contributions

This dissertation has presented three major studies: EV3RT, FMP-MC and ESPROF. In these studies, some important open issues around software platforms for modern embedded systems are discussed and possible solutions to them are proposed.

EV3RT is a real-time software platform for Mindstorms EV3 robotics kit. It is explained as an example to discuss the methodology of building a reliable RTOS-based platform for connected devices. As performance evaluation results indicate, advantages like performance predictability, low overhead and small footprint make RTOS more suitable for resource- or time-critical embedded systems than Linux. We have shown that the platform implementation effort can be significantly reduced by adopting open-source components strategically. The protection functionalities of HRP2 kernel allow us to design policies to detect and isolate potential defects in user application. These techniques are important for improving the reliability of a complex system. A generic dynamic module loading mechanism is proposed and used by EV3RT to support rebootless application updating. This mechanism can increase the productivity while keeping the simplicity and reliability of static OS design.

FMP-MC provides a testbed for investigating the scalability of traditional multi-core RTOS when running high-performance parallel applications on a 72-core processor. Several bottlenecks (contented spinlock, unoptimized cache locality, high-overhead thread-local storage, non-scalable allocator and slow mathematical library) commonly existing in current RTOS-based platforms have been identified by comparing FMP-MC with Linux. Methods to avoid them are proposed and evaluated with microbenchmarks. The scalability of optimized FMP-MC is very similar to Linux in realistic parallel workloads from PARSEC benchmark suite. In many cases, RTOS can deliver better absolute performance than Linux because of lower overhead in memory management and file operations. It suggests that, at least

for embedded many-core processors with tens of cores, traditional multi-core RTOS-based platforms with proper optimization can allow high-performance applications to scale well.

Existing monolithic profiling tools for high-performance applications cannot meet the requirements of diverse embedded systems, especially in terms of resource usage and measurement accuracy. Meanwhile, developing multiple small tools from scratch will severely limit the productivity. We have analyzed the requirements and common functionalities of different profiling tools. ESPROF, a generic infrastructure to support creating source-level profiling tools for multi/many-core embedded systems, is then proposed. Unlike rule-based infrastructures, it allows user to effortlessly implement advanced algorithms by full-featured programming languages (e.g. Python). As a proof of concept, a scalable call graph profiler named *ecg* is developed. Its algorithms generate optimized measurement code by exploiting the flexibility of ESPROF. User can analyze the profiling results with advanced HPC-oriented visualizer because *ecg* produces compatible data format. In measuring benchmark application on a 36-core platform, *ecg* shows much higher accuracy with smaller overhead compared to existing tool. We believe that an infrastructure like ESPROF can be very helpful for developing high-performance embedded applications.

Besides those addressed in the studies above, there are still many challenges ahead to improve the productivity, reliability and scalability of software platforms for embedded systems. All these contributions are open-source [43, 76, 75] and can provide some conceptual and practical bases for future research.

5.2 State of the Art and Future Trends

Reliability inside the Platform

EV3RT has discussed how to protect the platform from defected user-space applications. Some recent RTOS-based platforms (e.g. SafeRTOS[139], Zephyr OS [80]) also use static design with protection functionalities for high reliability. Meanwhile, components in the platform itself, such as kernel-mode device drivers, may also be vulnerable [36]. As the complexity of embedded systems increases, improving the reliability inside the platform is becoming another critical issue.

One approach to make the platform more reliable and secure is to move device drivers into user space by using a microkernel [121]. It can also significantly reduce the TCB (trusted computing base) size of the kernel. However, the increased overhead can be a potential issue for time-critical applications.

Another approach is to use an extra TEE (Trusted Execution Environment) for critical services [107]. The processor must provide a mechanism (e.g. ARM TrustZone[100]) to create an isolated environment that can only be accessed from the normal kernel in a secure way. The services and confidential data inside TEE can be protected even if the platform is compromised. This approach is more for security purposes rather than fault tolerance. Currently, only several processor families (e.g. x86, ARM) have hardware support for TEE.

A hypervisor to run multiple OSes with different levels of criticality on the same system can also improve reliability. It requires hardware virtualization assistance for efficient sharing and isolation of resources. The multi-OS approach means large memory footprint and CPU usage, and thus is mainly targeting high-end embedded systems. For low-cost devices, lightweight software-centric techniques (e.g. OS-level virtualization, time partitioning) seem promising. Although marketed as "TEE", MultiZone for RISC-V is a typical example of lightweight separation solution without specialized hardware [57].

Software Implementation Correctness

Reliability-related techniques depend heavily on the correctness of software implementation, which is a challenging topic with many unresolved issues.

Formal verification uses formal methods of mathematics to prove the correctness. seL4[65] and CertiKOS[55] are two examples of verified OS kernels. This approach requires significant manual effort and thus is only feasible to software with a small TCB at present. One major reason is that verification usually uses high-level abstractions but system software has low-level implementation. An attempt to address this issue is DeepSEA [117].

Alternatively, some new languages are proposed for writing programs with fewer bugs. For example, Rust [86] and Zig [62] aim to enforce safety with minimal performance penalty. Nevertheless, compiler bugs and optimizations can also introduce many vulnerabilities [39, 141]. It is vital (yet difficult) to ensure the correctness of compiler. CompCert is a project to create a high-assurance C compiler for critical embedded software [68].

Further, even if the software has proper functional behavior, the microarchitectural features of processor may still be exploited for attacking [48, 16]. To correctly implement the software, platform developers should also be aware of those hardware characteristics.

High Performance with Accelerators

The study of FMP-MC focuses on the scalability of CPU-intensive workloads. For some applications (e.g. image processing, AI inference), specialized hardware accelerators can provide better speedup with reduced power consumption. Guaranteeing the requirements

like response time and energy efficiency has become a major issue for systems including both CPU and accelerator workloads.

An accelerator can be treated as a shared resource or a heterogeneous core. Researchers are using self-suspension task models to analyze the schedulability of accelerator offloading but some important problems remain unresolved [30]. Project HERCULES have discussed many issues about predictable performance and energy efficiency with accelerators [134]. Besides, most existing performance analysis tools evaluate CPU and accelerators separately. They should be improved to provide better support for the heterogeneous architecture [101].

References

[1] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs, *Concurrency and Computation: Practice and Experience*, Vol. 22, No. 6(2009), pp. 685–701.

[2] AlDanial: Count Lines of Code (CLOC), https://github.com/AlDanial/cloc.

[3] Altium Ltd.: TASKING Embedded Profiler, https://www.tasking.com/products/tasking-embedded-profiler.

[4] Amazon Web Services, Inc.: The FreeRTOS Kernel, https://www.freertos.org/, 2019.

[5] Ammons, G., Ball, T., and Larus, J. R.: Exploiting hardware performance counters with flow and context sensitive profiling, *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation - PLDI '97*, ACM Press, 1997.

[6] Anderson, T.: The performance of spin lock alternatives for shared-money multiprocessors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1(1990), pp. 6–16.

[7] ARM Ltd.: Analyzing the performance of RTOS-based systems using Streamline. https://community.arm.com/tools/b/blog/posts/analyzing-rtos-based-systems-performance-using-streamline.

[8] ARM Ltd.: ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406-.

[9] ARM Ltd.: DS-5 Debugger, https://developer.arm.com/products/software-development-tools/ds-5-development-studio/ds-5-debugger/trace.

[10] ARM Ltd.: ELF for the ARM Architecture, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044f/IHI0044F_aaelf.pdf.

[11] ARM Ltd.: GNU Arm Embedded Toolchain, https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm.

[12] Auslander, M., Edelsohn, D., Krieger, O., Rosenburg, B., and Wisniewski, R.: Enhancement to the MCS lock for increased functionality and improved programmability, 2002.

[13] Azumi, T., Yamamoto, M., Kominami, Y., Takagi, N., Oyama, H., and Takada, H.: A New Specification of Software Components for Embedded Systems, *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, IEEE, may 2007.

[14] Baker, A.: *Windows NT Device Driver Book: A Guide for Programmers, with Disk with Cdrom*, Prentice Hall PTR, 1996.

[15] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A.: The multikernel, *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, ACM Press, 2009.

[16] Bechtel, M. and Yun, H.: Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention, *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, apr 2019.

[17] Bedi, P., Qayumi, K., and Kaur, T.: Home security surveillance system using multi-robot system, *International Journal of Computer Applications in Technology*, Vol. 45, No. 4(2012), pp. 272.

[18] Bell, R., Malony, A. D., and Shende, S.: ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis, *Euro-Par 2003 Parallel Processing*, Springer Berlin Heidelberg, 2003, pp. 17–26.

[19] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R.: Hoard, *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems - ASPLOS-IX*, ACM Press, 2000.

[20] Bienia, C., Kumar, S., Singh, J. P., and Li, K.: The PARSEC benchmark suite, *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*, ACM Press, 2008.

[21] BlueKitchen: BTstack, http://btstack.org.

[22] Bonwick, J.: The Slab Allocator: An Object-Caching Kernel Memory Allocator, 1994.

[23] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, M. F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y.-h., et al.: Corey: An Operating System for Many Cores, *OSDI*, 2008.

[24] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R. T., and Zeldovich, N.: An Analysis of Linux Scalability to Many Cores, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, Arpaci-Dusseau, R. H. and Chen, B.(eds.), USENIX Association, 2010, pp. 1–16.

[25] Boyd-wickizer, S., Kaashoek, M. F., Morris, R., and Zeldovich, N.: Non-scalable locks are dangerous.

[26] Bradley, P. J., de la Puente, J. A., Zamorano, J., and Brosnan, D.: A Platform for Real-Time Control Education with LEGO MINDSTORMS®1, *IFAC Proceedings Volumes*, Vol. 45, No. 11(2012), pp. 112–117.

[27] Bunzel, S.: AUTOSAR – the Standardized Software Architecture, *Informatik-Spektrum*, Vol. 34, No. 1(2010), pp. 79–83.

[28] Castello, A., Pena, A. J., Seo, S., Mayo, R., Balaji, P., and Quintana-Orti, E. S.: A Review of Lightweight Thread Approaches for High Performance Computing, *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, sep 2016.

[29] ChaN: FatFS, http://elm-chan.org/fsw/ff/00index_e.html.

[30] Chen, J.-J., Nelissen, G., Huang, W.-H., Yang, M., Brandenburg, B., Bletsas, K., Liu, C., Richard, P., Ridouard, F., Audsley, N., Rajkumar, R., de Niz, D., and von der Brüggen, G.: Many suspensions, many problems: a review of self-suspending tasks in real-time systems, *Real-Time Systems*, Vol. 55, No. 1(2018), pp. 144–207.

[31] Chikamasa, T.: nxtOSEK/JSP, http://lejos-osek.sourceforge.net.

[32] Chittimalli, P. K. and Shah, V.: GEMS: A Generic Model Based Source Code Instrumentation Framework, *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE, apr 2012.

[33] Clements, A. A., Almakhdhub, N. S., Saab, K. S., Srivastava, P., Koo, J., Bagchi, S., and Payer, M.: Protecting Bare-Metal Embedded Systems with Privilege Overlays, *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, may 2017.

[34] Clements, A. T., Kaashoek, M. F., and Zeldovich, N.: Scalable address spaces using RCU balanced trees, *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*, ACM Press, 2012.

[35] Coady, Y., Kiczales, G., Feeley, M., and Smolyn, G.: Using aspectC to improve the modularity of path-specific customization in operating system code, *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-9*, ACM Press, 2001.

[36] CVE Details: Linux Kernel Vulnerability Statistics, https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[37] Dagum, L. and Menon, R.: OpenMP: an industry standard API for shared-memory programming, *IEEE Computational Science and Engineering*, Vol. 5, No. 1(1998), pp. 46–55.

[38] DENX Software Engineering: Das U-Boot – the Universal Boot Loader, http://www.denx.de/wiki/U-Boot.

[39] D'Silva, V., Payer, M., and Song, D.: The Correctness-Security Gap in Compiler Optimization, *2015 IEEE Security and Privacy Workshops*, IEEE, may 2015.

[40] Edison Design Group Inc.: The C++ Front End, https://www.edg.com/c.

[41] eSOL: eBinder, https://www.esol.co.jp/embedded/ebinder2.html.

[42] ET Robocon Executive Committee: Reference construction of EV3way-ET robot, https://github.com/ETrobocon/etroboEV3/wiki.

[43] EV3RT: English home page, http://ev3rt-git.github.io.

[44] EV3RT: Japanese home page, http://dev.toppers.jp/trac_user/ev3pf/wiki/WhatsEV3RT.

[45] Ferrari, M. and Ferrari, G.: *Building robots with LEGO Mindstorms NXT*, Syngress, 2011.

[46] Fineberg, S. A.: Using MPI-Portable Parallel Programming with the Message-Passing Interface, by William Gropp, *Scientific Programming*, Vol. 5, No. 3(1996), pp. 275–276.

[47] Gaska, T., Watkin, C., and Chen, Y.: Integrated Modular Avionics - Past, present, and future, *IEEE Aerospace and Electronic Systems Magazine*, Vol. 30, No. 9(2015), pp. 12–23.

[48] Ge, Q., Yarom, Y., Cock, D., and Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware, *Journal of Cryptographic Engineering*, Vol. 8, No. 1(2016), pp. 1–27.

[49] Geimer, M., Shende, S. S., Malony, A. D., and Wolf, F.: A Generic and Configurable Source-Code Instrumentation Component, *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2009, pp. 696–705.

[50] Geimer, M., Wolf, F., Wylie, B. J. N., Ábrahám, E., Becker, D., and Mohr, B.: The Scalasca performance toolset architecture, *Concurrency and Computation: Practice and Experience*, Vol. 22, No. 6(2010), pp. 702–719.

[51] Gerndt, M., César, E., and Benkner, S.: *Automatic Tuning of HPC Applications-The Periscope Tuning Framework (PTF)*, Shaker Verlag, 2015.

[52] GNU Project: The GNU C Library, https://www.gnu.org/software/libc/libc.html.

[53] GNU Project: GNU Compiler Collection, https://gcc.gnu.org.

[54] Green Hills Software: TimeMachine Debugging Suite, https://www.ghs.com/products/timemachine.html.

[55] Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V., and Costanzo, D.: CertiKOS: An Extensible Architecture for Building Certified Concurrent {OS} Kernels, *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 653–669.

[56] HaWe: HaWe Brickbench Benchmark Test, http://www.mindstormsforum.de/viewtopic.php?f=71&t=8095.

[57] Hex Five Security, Inc.: MultiZone, https://hex-five.com/multizone-security-sdk/.

[58] Hoffmann, M., Dietrich, C., and Lohmann, D.: Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience, *2nd GI W'shop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, 2013.

[59] Japan Aerospace Exploration Agency: High-Reliability Real-time Operating System, http://rtos.jaxa.jp.

[60] Japan Embedded System Technology Association: Embedded Technology Software Design Robot Contest, http://www.etrobo.jp.

[61] KALRAY Corporation: MPPA: The Supercomputing on a chip solution, http://www.kalrayinc.com/kalray/products/.

[62] Kelley, A.: The Zig Programming Language, https://ziglang.org/.

[63] Kerrisk, M.: sched - overview of CPU scheduling, http://man7.org/linux/man-pages/man7/sched.7.html.

[64] Klassner, F.: A case study of LEGO Mindstorms suitability for artificial intelligence and robotics courses at the college level, *ACM SIGCSE Bulletin*, Vol. 34, No. 1(2002), pp. 8.

[65] Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., and Heiser, G.: Comprehensive formal verification of an OS microkernel, *ACM Transactions on Computer Systems*, Vol. 32, No. 1(2014), pp. 1–70.

[66] Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M. S., and Nagel, W. E.: The Vampir Performance Analysis Tool-Set, *Tools for High Performance Computing*, Springer Berlin Heidelberg, 2008, pp. 139–155.

[67] Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., and Wolf, F.: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir, *Tools for High Performance Computing 2011*, Springer Berlin Heidelberg, 2012, pp. 79–91.

[68] Krebbers, R., Leroy, X., and Wiedijk, F.: Formal C Semantics: CompCert and the C Standard, *Interactive Theorem Proving*, Springer International Publishing, 2014, pp. 543–548.

[69] Larson, P.-Å. and Krishnan, M.: Memory allocation for long-running server applications, *Proceedings of the first international symposium on Memory management - ISMM '98*, ACM Press, 1998.

[70] Lea, D. and Gloger, W.: A memory allocator, 1996.

[71] Lee, T.-H.: Real-Time Face Detection and Recognition on LEGO Mindstorms NXT Robot, *Advances in Biometrics*, Springer Berlin Heidelberg, 2007, pp. 1006–1015.

[72] LEGO Group: LEGO MINDSTORMS EV3 source code, https://github.com/mindboards/ev3sources.

[73] LEGO Group: MINDSTORMS robot kit, https://www.lego.com/en-us/themes/mindstorms/about.

[74] Levine, J. R.: *Linkers and Loaders*, Morgan Kaufmann, 2000.

[75] Li, Y.: Prototype implementation of esprof and ecg, https://github.com/es-prof.

[76] Li, Y.: TOPPERS/FMP on TILE-Gx Many Core Processor, https://github.com/fmp-mc.

[77] Li, Y., Ishikawa, T., Matsubara, Y., and Takada, H.: A platform for LEGO mindstorms EV3 based on an RTOS with MMU support, *OSPERT 2014*, 2014.

[78] Lindlan, K., Cuny, J., Malony, A., Shende, S., Mohr, B., Rivenburgh, R., and Rasmussen, C.: A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates, *ACM/IEEE SC 2000 Conference (SC'00)*, IEEE, 2000.

[79] Lingegowda, V.: Optimization of Data Read/Write in a Parallel Application, https://software.intel.com/en-us/blogs/2013/03/04/optimization-of-data-readwrite-in-a-parallel-application.

[80] Linux Foundation: Zephyr OS, https://www.zephyrproject.org.

[81] Linux Kernel Organization, Inc.: Linux Kernel Documentation, https://www.kernel.org/doc/htmldocs.

[82] Linux Programmer's Manual: syscalls - Linux system calls, http://man7.org/linux/man-pages/man2/syscalls.2.html.

[83] Linux.com: What's New in Linux 2.6.39: Ding Dong, the Big Kernel Lock is Dead, https://www.linux.com/learn/whats-new-linux-2639-ding-dong-big-kernel-lock-dead.

[84] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A.: The Linux scheduler, *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16*, ACM Press, 2016.

[85] Masmano, M., Ripoll, I., Crespo, A., and Real, J.: TLSF: a new dynamic memory allocator for real-time systems, *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, IEEE, 2004.

[86] Matsakis, N. D. and Klock, F. S.: The rust language, *ACM SIGAda Ada Letters*, Vol. 34, No. 3(2014), pp. 103–104.

[87] Matz, M., Hubicka, J., Jaeger, A., and Mitchell, M.: System V Application Binary Interface, *AMD64 Architecture Processor Supplement*, No. 1(2013), pp. 1.

[88] Mellanox Technologies: Overview of the 72-core TILE-Gx72 processor, http://www.mellanox.com/page/products_dyn?product_family=238&mtag=tile_gx72.

[89] Mellanox Technologies: TILEncore-Gx72 Intelligent Application Adapter, http://www.mellanox.com/page/products_dyn?product_family=231.

[90] National Instruments: Laboratory Virtual Instrument Engineering Workbench (LabVIEW), http://www.ni.com/labview.

[91] Navarro, A., Asenjo, R., Tabik, S., and Cascaval, C.: Analytical Modeling of Pipeline Parallelism, *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, IEEE, sep 2009.

[92] Nethercote, N.: Dynamic binary analysis and instrumentation, Technical report, University of Cambridge, Computer Laboratory, 2004.

[93] Olofsson, A., Nordstrom, T., and Ul-Abdin, Z.: Kickstarting high-performance energy-efficient manycore architectures with Epiphany, *2014 48th Asilomar Conference on Signals, Systems and Computers*, IEEE, nov 2014.

[94] Ozkaya, I.: If it does not scale, it does not work!, *IEEE Software*, Vol. 36, No. 2(2019), pp. 4–7.

[95] PARSEC team: The PARSEC Benchmark Suite, http://parsec.cs.princeton.edu.

[96] Percepio AB.: Percepio Tracealyzer, https://percepio.com/tracealyzer/.

[97] Perret, Q., Maurère, P., Noulard, É., Pagetti, C., Sainrat, P., and Triquet, B.: Mapping hard real-time applications on many-core processors, *Proceedings of the 24th International Conference on Real-Time Networks and Systems - RTNS '16*, ACM Press, 2016.

[98] Pinho, L. M., Nélis, V., Yomsi, P. M., Quiñones, E., Bertogna, M., Burgio, P., Marongiu, A., Scordino, C., Gai, P., Ramponi, M., and Mardiak, M.: P-SOCRATES: A parallel software framework for time-critical many-core systems, *Microprocessors and Microsystems*, Vol. 39, No. 8(2015), pp. 1190–1203.

[99] Pinho, L. M., Quinones, E., Bertogna, M., Marongiu, A., Nélis, V., Gai, P., and Sancho, J.: High-Performance and Time-Predictable Embedded Computing, *High-Performance and Time-Predictable Embedded Computing*, River Publisher, 2018, pp. 1–236.

[100] Pinto, S. and Santos, N.: Demystifying Arm TrustZone, *ACM Computing Surveys*, Vol. 51, No. 6(2019), pp. 1–36.

[101] Pinto, V. G., Schnorr, L. M., Stanisic, L., Legrand, A., Thibault, S., and Danjean, V.: A visual performance analysis framework for task-based parallel applications running on hybrid clusters, *Concurrency and Computation: Practice and Experience*, Vol. 30, No. 18(2018), pp. e4472.

[102] Quinlan, D. and Liao, C.: The rose source-to-source compiler infrastructure, *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Citeseer, 2011, pp. 1.

[103] Red Hat: Newlib, a C standard library for embedded systems, https://sourceware.org/newlib.

[104] Rosinski, J.: General Purpose Timing Library, https://github.com/jmrosinski/GPTL.

[105] Roth, M., Best, M. J., Mustard, C., and Fedorova, A.: Deconstructing the overhead in parallel applications, *2012 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, nov 2012.

[106] Rusling, D. A.: The Linux Kernel: Memory Management, http://www.tldp.org/LDP/tlk/mm/memory.html.

[107] Sabt, M., Achemlal, M., and Bouabdallah, A.: Trusted Execution Environment: What It is, and What It is Not, *2015 IEEE Trustcom/BigDataSE/ISPA*, IEEE, aug 2015.

[108] Saidi, S., Ernst, R., Uhrig, S., Theiling, H., and de Dinechin, B. D.: The shift to multicores in real-time and safety-critical systems, *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, IEEE, oct 2015.

[109] Salzman, P. J., Burian, M., and Pomerantz, O.: The Linux kernel module programming guide, http://tldp.org/LDP/lkmpg/2.4/html/book1.htm.

[110] Sangiovanni-Vincentelli, A. and Martin, G.: Platform-based design and software design methodology for embedded systems, *IEEE Design & Test of Computers*, Vol. 18, No. 6(2001), pp. 23–33.

[111] Schwarz, B., Debray, S., and Andrews, G.: Disassembly of executable code revisited, *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, IEEE Comput. Soc, 2002.

[112] Seeed Technology Co.,Ltd.: Air602 WiFi Module, https://www.seeedstudio.com/Air602-WiFi-Module.html.

[113] Shalloway, A. and Trott, J. R.: *Design patterns explained: a new perspective on object-oriented design*, Pearson Education, 2004.

[114] Shende, S. S. and Malony, A. D.: The Tau Parallel Performance System, *The International Journal of High Performance Computing Applications*, Vol. 20, No. 2(2006), pp. 287–311.

[115] Singh, A. K., Shafique, M., Kumar, A., and Henkel, J.: Mapping on multi/many-core systems, *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*, ACM Press, 2013.

[116] Sinha, S., Koedam, M., van Wijk, R., Nelson, A., Nejad, A. B., Geilen, M., and Goossens, K.: Composable and Predictable Dynamic Loading for Time-Critical Partitioned Systems, *2014 17th Euromicro Conference on Digital System Design*, IEEE, aug 2014.

[117] Sjöberg, V., Sang, Y., Weng, S.-c., and Shao, Z.: DeepSEA: A Language for Certified System Software, http://flint.cs.yale.edu/certikos/publications/deepsea19.html.

[118] Spinczyk, O. and Lohmann, D.: The design and implementation of AspectC++, *Knowledge-Based Systems*, Vol. 20, No. 7(2007), pp. 636–651.

[119] Sysprogs: Using VisualGDB FreeRTOS Tracing to Optimize Real-time Code, https://visualgdb.com/tutorials/profiler/realtime/freertos/.

[120] Søborg, A.: MonoBrick EV3 Firmware, http://www.monobrick.dk.

[121] Tanenbaum, A., Herder, J., and Bos, H.: Can We Make Operating Systems Reliable and Secure?, *Computer*, Vol. 39, No. 5(2006), pp. 44–51.

[122] Texas Instruments Inc.: Starterware for AM1808 SoC, http://processors.wiki.ti.com/index.php/StarterWare.

[123] The leJOS project: leJOS EV3, http://www.lejos.org.

[124] The Robocup Federation: RoboCup Junior, https://junior.robocup.org/.

[125] Tomiyama, H., Honda, S., and Takada, H.: Real-time operating systems for multicore embedded systems, *2008 International SoC Design Conference*, IEEE, nov 2008.

[126] TOPPERS Project Inc.: Introduction to the TOPPERS/FMP kernel, https://www.toppers.jp/en/fmp-kernel.html.

[127] TOPPERS Project Inc.: mruby on EV3RT + TECS, https://www.toppers.jp/tecs.html.

[128] TOPPERS Project Inc.: TOPPERS new generation kernel specification, http://www.toppers.jp/documents.html.

[129] TOPPERS Project Inc.: TOPPERS/HRP2 kernel, http://www.toppers.jp/en/hrp2-kernel.html.

[130] TOPPERS Project Inc.: TraceLogVisualizer, https://www.toppers.jp/tlv.html.

[131] TOPPERS Project Inc.: TOPPERS RTOS kernels, http://www.toppers.jp/documents.html, 2019.

[132] TRON Forum: ITRON Specification, http://www.tron.org/specifications.

[133] Ungerer, T., Bradatsch, C., Gerdes, M., Kluge, F., Jahr, R., Mische, J., Fernandes, J., Zaykov, P., Petrov, Z., Boddeker, B., Kehr, S., Regler, H., Hugl, A., Rochange, C., Ozaktas, H., Casse, H., Bonenfant, A., Sainrat, P., Broster, I., Lay, N., George, D., Quinones, E., Panic, M., Abella, J., Cazorla, F., Uhrig, S., Rohde, M., and Pyka, A.: parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability, *2013 Euromicro Conference on Digital System Design*, IEEE, sep 2013.

[134] University of Modena and Reggio Emilia: Project HERCULES, http://hercules2020.eu/index.php/dissemination-2/publications/.

[135] Valk, L.: *LEGO MINDSTORMS EV3 Discovery Book: A Beginner's Guide to Building and Programming Robots*, No Starch Press, 2014.

[136] Wahyudi, S.: Real-Time Control System for a Two-Wheeled Inverted Pendulum Mobile Robot, *Advanced Knowledge Application in Practice*, Sciyo, nov 2010.

[137] Wang, Z., Sanchez, A., and Herkersdorf, A.: SciSim: a software performance estimation framework using source code instrumentation, *Proceedings of the 7th international workshop on Software and performance - WOSP '08*, ACM Press, 2008.

[138] Wentzlaff, D. and Agarwal, A.: Factored operating systems (fos), *ACM SIGOPS Operating Systems Review*, Vol. 43, No. 2(2009), pp. 76.

[139] WITTENSTEIN Group: SAFERTOS, https://www.highintegritysystems.com/safertos/.

[140] WRO Advisory Council (AC): World Robot Olympiad, http://www.wroboto.org.

[141] Yang, X., Chen, Y., Eide, E., and Regehr, J.: Finding and understanding bugs in C compilers, *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*, ACM Press, 2011.

[142] Ylonen, T. and Lonvick, C.: The secure shell (SSH) authentication protocol, http://tools.ietf.org/html/rfc4252.

Appendix A

List of Related Publications

Journal Papers

[1] Yixiao Li, Yutaka Matsubara, Hiroaki Takada, "EV3RT: A Real-time Software Platform for LEGO Mindstorms EV3", Computer Software, Vol.34 No.4, p. 91-115, 2017.

[2] Yixiao Li, Yutaka Matsubara, Hiroaki Takada, "A Comparative Analysis of RTOS and Linux Scalability on an Embedded Many-core Processor", Journal of Information Processing, Vol.26, p. 225-236, 2018.

[3] Yixiao Li, Yutaka Matsubara, Hiroaki Takada, "esprof: A Generic Profiling Infrastructure for Multi/Many-Core Embedded Systems", Computer Software, Vol.37 No.1, 2020. (to appear)

International Conferences (Refereed)

[1] Yixiao Li, Takuya Ishikawa, Yutaka Matsubara, Hiroaki Takada, "A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support", 10th annual workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2014), July 2014.