A UNIFIED RUN-TIME UPDATING AND TASK MIGRATION MECHANISM

Wictor Lund
ABSTRACT

Run-time updating is relevant in systems where a restart of the whole system is expensive. It is desirable to have the possibility to safely update or reconfigure a part of the system at a low cost. Task migration is the act of transferring a process between two processing units. Task migration becomes more and more essential as an operating system feature as the on-chip core-count increases as current operating system structure will not scale with core-count ad infinitum. If run-time updating and task migration are addressed by extracting the tasks state from the operating system, it is possible to create an unified implementation, where a tasks state is extracted and processed accordingly.

This thesis introduces the notions needed for task state extraction, and a implementation of both run-time updating and task migration is presented. The presented implementation is based on FreeRTOS running on a multi-core architecture with shared memory. The run-time updating scenario is intended to run in a non-critical part in a mixed-critical environment.

Keywords: task migration, run-time updating, task state extraction
CONTENTS

Abstract i

Contents ii

List of Figures iv

Glossary vi

1 Introduction 1
  1.1 RECOMP project . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.2 Problem statement . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  1.3 Thesis structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 Platform Model 3
  2.1 Task Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
  2.2 Mapping the model onto different operating systems . . . . . . . . 5
  2.3 Hardware platforms considerations . . . . . . . . . . . . . . . . . . 7
  2.4 Chosen platform . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.5 Task migration scenarios . . . . . . . . . . . . . . . . . . . . . . . . 13

3 Run-time updating and Task migration methodologies 21
  3.1 Description of the migration process . . . . . . . . . . . . . . . . . . 23
  3.2 Run-time updating methodology . . . . . . . . . . . . . . . . . . . . 24
  3.3 Task migration methodology . . . . . . . . . . . . . . . . . . . . . . 25
    3.3.1 Shared memory migration . . . . . . . . . . . . . . . . . . . . 27
    3.3.2 Distributed memory migration . . . . . . . . . . . . . . . . . . 28

4 Implementation 30
  4.1 Platform . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
  4.2 Implementation overview . . . . . . . . . . . . . . . . . . . . . . . . 30
    4.2.1 Binaries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
    4.2.2 Directory hierarchy . . . . . . . . . . . . . . . . . . . . . . . . 31
  4.3 Code used from external sources . . . . . . . . . . . . . . . . . . . . 32
4.3.1 Niels Provos <sys/tree.h> implementation from the OpenBSD project
4.3.2 <sys/queue.h> implementation from the OpenBSD project
4.3.3 Libdwarf

4.4 Task manager
4.4.1 Executable and Linkable Format (ELF)
4.4.2 Functionality
4.4.3 The application binary
4.4.4 Methods
4.4.5 Task start-up process

4.5 Linker
4.5.1 Methods

4.6 Migrator
4.6.1 Functionality

4.7 Pointer tracer
4.7.1 High-level description
4.7.2 Implementation details
4.7.3 Dwarf
4.7.4 Dwarfif
4.7.5 Low-level description
4.7.6 Low-level algorithm

4.8 Build system
4.8.1 Ninja build system
4.8.2 Structure of the resulting binary

4.9 Running FreeRTOS on multi-core
4.9.1 SMP Multicore with separate kernels on each core
4.9.2 Boot process

4.10 Use-cases
4.10.1 Run-time updating
4.10.2 Task migration

5 Future work and conclusions
5.1 Conclusions
5.2 Future work

Bibliography

Swedish summary

A Appendix
A.1 Run-time updating pseudo-example
A.2 Task migration pseudo-example
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Non-uniform memory access (NUMA), shared memory system.</td>
</tr>
<tr>
<td>2.2</td>
<td>Distributed memory system.</td>
</tr>
<tr>
<td>2.3</td>
<td>Block diagram of the Versatile Express [1]</td>
</tr>
<tr>
<td>2.4</td>
<td>Possible task migration scenarios for many-core architectures</td>
</tr>
<tr>
<td>2.5</td>
<td>An overview of the memories in a Cortex-A9 MPCore [2] relevant to task migration scenario 1.</td>
</tr>
<tr>
<td>2.6</td>
<td>An overview of the memory in a Tilera TilePro [3]-like network-on-chip architecture.</td>
</tr>
<tr>
<td>2.7</td>
<td>An overview of the processors and the MMU’s in the Raspberry Pi.</td>
</tr>
<tr>
<td>3.1</td>
<td>Illustration of the migration process.</td>
</tr>
<tr>
<td>3.2</td>
<td>Illustration of the run-time updating process, viewing change of the tasks content with time. This is the process described in Figure 3.1 when applied on run-time updating.</td>
</tr>
<tr>
<td>3.3</td>
<td>Figure showing the data sections in a binary with a programmer defined «rtu data» section, specifying the run-time updatable state.</td>
</tr>
<tr>
<td>3.4</td>
<td>Illustration of task migration scenario 1., viewing change of the tasks content with time. This is the process described in Figure 3.1 when applied on task migration scenario 1., as described in section 2.5.</td>
</tr>
<tr>
<td>4.1</td>
<td>Illustration of dependencies between the software modules in the system.</td>
</tr>
<tr>
<td>4.2</td>
<td>Illustration the linking- and execution view in ELF [4].</td>
</tr>
<tr>
<td>4.3</td>
<td>Petri net describing the state flow of a task managed by the task manager. The Petri net shown here is in it’s initial state.</td>
</tr>
<tr>
<td>4.4</td>
<td>Illustration of actors involved in a run-time updating or task migration process.</td>
</tr>
<tr>
<td>4.5</td>
<td>Illustration of memory layout graph with a pointer to an integer array.</td>
</tr>
<tr>
<td>4.6</td>
<td>Illustration of memory layout graphs with pointers pointing to valid and invalid memory objects.</td>
</tr>
<tr>
<td>4.7</td>
<td>Illustration of memory layout graphs with a pointer to an array of pointers.</td>
</tr>
<tr>
<td>4.8</td>
<td>An illustration of how the variable const int *var; defined in the global scope, would be described with the dwarf format.</td>
</tr>
<tr>
<td>4.9</td>
<td>Illustration of causality dependencies in the life-time of built software.</td>
</tr>
</tbody>
</table>
4.10 Illustration of both virtual and physical memory layout. . . . . . . . . 58
GLOSSARY

ABI  Application Binary Interface. Specifies an interface between binary entities in an application, e.g. how arguments are passed between functions. The ABI is dependent on the ISA, i.e. one ISA can have several ABI. An ABI is not generally compatible between different ISA.

AMP  Asymmetric multi-processing. A multi-processing system where the operating system view is not the same for every processing unit is said to be AMP.

checkpoint  A point in the execution of a program which meet a certain condition. Example: A checkpoint can be a point in the execution of a program where it is safe to write the state of the program to a persistent storage, so that the execution can continue from this point at a later stage if for example a failure occur.

CPO  CheckPointable Object. A object made from a task in a checkpointable state. This term is defined in Chapter 3.

GPGPU  General Purpose Graphical Processing Unit. A processing unit used for rendering graphics, but which can also perform more generic computations.

GPU  Graphical Processing Unit. A processing unit used for rendering graphics.

ISA  Instruction Set Architecture. The part of a computer architecture describing a computers instruction set, registers, data types and memory view.

mixed-critical  An application is said to have mixed-criticality if the application contains both
critical and non-critical elements. This term is often used in association with safety-critical applications.

**MMU**
Memory Management Unit, provides virtual memory.

**MPU**
Memory Protection Unit, provides memory protection.

**non-critical**
An element in an application is said to be non-critical if the element is critical to the functionality or safety of the whole application.

**position independent binary**
A binary is position independent if it is possible to execute the code in it, independent on what memory address the binary is located on, i.e. the binary cannot use absolute addressing to access internal code or data.

**relocatable**
A memory region is said to be relocatable if it can be moved to another address in the address space without losing its functionality.

**SMP**
Symmetric multi-processing. A multi-processing system where the operating system view is the same for every processing unit is said to be SMP.

**SoC** System on Chip. A chip containing all components in a computer system, thus if using a SoC very few external components are used for building a system capable of computing.
1 INTRODUCTION

In embedded systems design, software implementation provides more flexibility than hardware implementations, trading off for performance and energy consumption. Flexibility can be increased by using the right tools. The thesis introduces methodologies for run-time updating and task migration to support the design process for embedded systems. The run-time updating methodology presented in the thesis offers a method for updating software in an embedded system without restarting the system. To further improve flexibility the thesis introduces a task migration methodology, which can help in the design of embedded system where a dynamic task mapping is desired.

The run-time updating and task migration methodologies, although seemingly different, are unified as a methodology for software flexibility and robustness. The potential of the task migration and run-time updating methodologies is demonstrated in the form of a unified implementation. The run-time updating functionality provides a way to, while the system is online, update non-critical software in mixed-critical solutions.

1.1 RECOMP project

The thesis was done within the RECOMP project. The RECOMP [5] (REduced certification COst for trusted Multi-core Platforms) research project will establish methods, tools and platforms for enabling cost-efficient certification and re-certification of mixed-critical systems. RECOMP will provide reference design and platform architectures together with the required design method and tools for achieving the stated goal. The thesis reviews material that will lay the foundation of implementing operating system support for safe multi-core integration and core-to-core communication. The project is an international ARTEMIS joint undertaking project and is active in the years 2010 to 2013.
1.2 Problem statement

The society becomes more dependent on software in high-availability systems. At the same time increased software complexity makes the need for up-to-date software more essential. The high dependency combined with increased complexity makes updating of critical high-availability systems very expensive. The increased complexity also raises the software updating frequency, which in turn increase the total cost of ownership. A solution to the problem is to minimize the cost of software updating by using run-time updating, and thus avoiding a expensive reboot of the system when an update takes place.

One demanding effort in the process of designing embedded real-time many-core systems is to map the tasks to cores. The most straightforward way of performing a task like that is to create a static mapping. However, when confronted with unpredictable loads, tight power budgets and huge time-slack, static mapping is not always enough to complete the task. Tasks can turn from having small CPU time requirements to start competing with other tasks on the same core. Another task can be loading a CPU core enough to occupy it entirely, just to let go of most of the CPU resources later, leaving the core to consume unnecessarily much power when the task wakes up for a short job. Task migration is presented as a solution to these problems. The solution is to move tasks from a over-loaded core to meet quality of service and real-time requirements, and on a under-loaded core move tasks away to let the core idle for longer periods and save power and energy.

1.3 Thesis structure

The thesis is divided into two parts. The first part defines the models used and the implemented functionality. The second part describes the implementation based on the results of the first part.

Chapter 2 describes a model of the platform describing the environment run-time updating and task migration being studied in. Chapter 3 describes run-time updating and task migration as methodologies and introduces notions needed for engineering the implementation, presented in Chapter 4. The implementation is presented as a set of software libraries providing flexibility in the form of the possibility to perform task migration and run-time updating, for embedded applications.
2 Platform Model

This chapter describes the environment in which the run-time updating and task migration methodologies in Chapter 3 are defined. This chapter also explains the platform on which the implementation in Chapter 4 is based on. The discussed platform model consists of a software and a hardware part. The software part consists mostly of the operating system running on the hardware platform. The model of the operating system is described as general as possible to avoid dependencies to specific software or hardware. The hardware platform restricts what software is able to run on it. The thesis focuses on many-core platforms that are likely to be popular in the future, e.g. platforms that could scale up to thousands of cores [6]. The chapter will begin by describing the software aspect of the platform model. After this the chapter will describe the hardware platforms that are taken into consideration when the model is defined. The hardware platform that the implementation is based on, is explained in further detail. Finally all possible task migration and run-time updating scenarios the platform model can be involved with are presented.

To gain practical experience during the development of the platform model the operating system FreeRTOS [7] has been adopted. FreeRTOS is a lightweight RTOS written with about 5000 lines of C code, the main components the operating system provides is a pre-emptive scheduler and inter-process communication facilities such as queues, semaphores and timers. The crucial difference between programming in FreeRTOS and programming without operating system is that the operating system has set up a tick interrupt able to pre-empt and transparently switch tasks, according to the tasks priorities. The operating system has the favourable property of having a very simple task model, mainly influenced by the notion of transparently switching between tasks, which concludes the operating system being close to ideal as a tool for developing new task models.
2.1 Task Model

The most important aspect of the software part of the platform model is the task model. Addressing both task migration and run-time updating requires the knowledge of what to migrate and update. This requirement constitutes a need for a task model describing the resources a task is consisting of.

The task state is the most central aspect of the task model. The discussed task state describes everything a task consists of. It is important that the task state is defined in a way allowing the task to be put into a state where it is possible to perform both run-time updating and task migration. It is desirable to know what resources are allocated for a task, and which of these resources are migratable. The task state consists of resources with different properties, for this reason the task state is divided into two parts, the user-state and the OS-state. The software part of the platform model consists mainly of a task state, which have to be divided into two parts to distinguish between resources with different properties.

The first part of the task state is the user-state. The user-state consists of the heap, the stack and code. These three entities exist in a linear address space and is directly accessible by the user. Code running in a context able to manipulate the user-state directly is in many operating systems said to be executing in user-land. Because the user-state exists in a linear address space it is easily migrated if the memory layout can be preserved.

The second part of the task state is the OS-state. The OS component of the state is not directly accessible, in contrast to the user-state. The reason for the OS-state not being directly accessible, is because it consists of resources allocated by the OS, e.g. files, network connections and inter-task communication facilities. The OS-state is accessed via some kind of API, opposed to the user-state. For example, in a POSIX-compliant operating system [8], the programmer can send a read() system call to the operating system, manipulating a file descriptor included in the operating system managed OS-state. The OS-state can be problematic to migrate, since the resources the component consists of are often bound to local hardware or to other resources which will or cannot be migrated.

The notion described here called the task state is the description of in what state the task in question is. The task state could essentially been seen as an object describing
exactly where in its execution a task is. Since the exact description of where in the execution a task is, in a general sense, the *task state* is usually large in size.

### 2.2 Mapping the model onto different operating systems

A new task model is better described in relation to existing task models, to show the differences. Every operating system has a model describing the tasks running on it. The most interesting aspect of the model is the *task state*, which is not always explicit in task models found in existing operating system implementations. The software part of the platform model will be put in its context by using various operating systems as examples.

Tanenbaum and Woodhull [9] presents a process model that is widely thought in operating system courses, their *process model* is also used in the implementation of Tanenbaums Minix operating system. In Tanenbaum and Woodhulls process model the term *core image* is used for describing the same entity as the *user-state* in this model. Tanenbaum and Woodhull further declare a *process state* describing if the process is in a running, ready or blocked state. This *process state* is, in this model, included in the *OS-state*. Tanenbaum and Woodhulls *process model* describes the de facto standard *process model* for most operating systems today.

FreeRTOS [7] is a lightweight RTOS used as the reference OS in this thesis. FreeRTOS has very few OS features compared to standard operating system kernels, accordingly the task model is comparatively primitive. Many more advanced operating system kernels provide virtual memory to the user programs, to offer a more dynamic memory layout and to protect user programs from corrupting them self and others. Virtual memory requires hardware support in the form of a MMU (Memory Management Unit), which is not always available on embedded systems. Since FreeRTOS tries to keep its footprint small, virtual memory is not offered as an operating system feature. However, FreeRTOS offers *memory protection* support [10], for the platforms incorporated with a MPU (Memory Protection Unit). This feature is currently not frequently used, as few platforms currently have an MPU. Since FreeRTOS does not usually have memory protection enabled, the *OS-state* technically exist in the
same visible memory space as the user-state, however, the OS-state is always accessed through an API. With memory protection enabled it is possible to only allow direct access to the user-state of the currently executing task, thus protecting user tasks from corrupting the operating systems internal state as well as the user- and OS-state of other tasks. The FreeRTOS task model has a process state, e.g a task can be running, ready or blocked, similar to the one Tanenbaum and Woodhull uses. In conclusion, the separation between OS-state and user-state is more blurry in FreeRTOS than in operating systems with clear privilege separation.

Linux has a process model largely based on Tanenbaum and Woodhulls model, but developed and modified to include and support features required for modern production operating systems. In Linux, a process or thread can allocate resources from the kernel. The definition of resource is kept loose to potentially include a growing variety of objects. As an example of such a resource can be given the frequently allocated resources memory regions and file descriptors. The user-process usually accesses the resources in the OS-state by performing software interrupts. This means that the user-process yields the CPU when it accesses the OS-state. The user-state is accessed directly when the user-process has control over the CPU. Another way to access the OS-state is to use the CPUs memory accessing instructions to cause kernel-traps. It should be mentioned that the available user-state is managed by the OS-state, since it keeps track of what memory regions are allocated. The Linux process model is probably the most used process model today and the internals of it is far too complex to be explained here.

It has been shown that the task model is relatable to existing task model by describing the task model in terms of other task models and vice versa. Three varying examples has been used, the academic Tanenbaum and Woodhull model, the simplistic FreeRTOS model and the practical Linux model, which shows variations in the models de facto used today. The platform model has in this section been explained in terms of how it handles tasks and processes, and we will go on to describe the hardware part of the platform model.
2.3 Hardware platforms considerations

Software is always, in the end, executed on a hardware platform. The hardware platform affects the software design, and the software design affects the choice of hardware. For the sake of investigating task migration the system architecture is required to be multiprocessing since task migration is always performed between two processing units. Run-time updating, on the other hand, can be performed on systems with only one processing unit. Run-time updating further requires that it is possible to write to the program memory at run-time. This section highlights architectures that are relevant for the investigated subjects, namely task migration and run-time updating. This section is the starting point for platform that is discussed in the rest of the thesis.

A shared memory architecture is a many-core architecture where the cores share a common memory. A shared memory architecture often have MMUs:s and support virtual memory. In embedded systems, however this is not always the case. Complex hardware such as a MMU with a lot of features and thus increased gate count, leads to increased failure rate and increased power consumption. This is the reason it is sometimes preferable to use a less complex MPU instead of a MMU. Another feature that many shared memory systems have is cache coherency. As example, the quad core Cortex-A9 MPCore [2] has a cache coherency protocol using snooping and the 64-core Tilera TilePro64 has distributed cache coherency protocol [3]. The shared memory architectures considered relevant in the thesis are on-chip many-core architectures where each core can have it’s own cache or scratch pad, and main memory is shared.

Shared memory systems have been the most popular many-core systems since they give a smooth transition from single-core system to many-core SMP systems, and gives thus better compatibility towards legacy code. However, some claim that cache coherency will not be feasible in the future [6], while others claim that cache coherency will scale to at least 512 cores [11]. The advantages and future prospects of shared memory architectures in SMP systems suggests that shared memory systems will continue to be used in the future.

A technique to make shared memory systems more scalable is NUMA, short for Non-Uniform Memory Access. The difference between NUMA systems and ordinary shared memory systems is letting memory accesses to different memory banks have different latency. A memory bank close to a given processor has lower access latency than a memory bank more far away from the given processor. Another perspective on
NUMA is to view certain parts of the system having private memory, where all private memories are shared and mapped in one single memory space. A NUMA system is illustrated in Figure 2.1. The non-uniformity in Figure 2.1 comes from the fact that if «core 1» performs a memory access to «memory bank 1», the memory access is done directly through it’s cross-bar switch, but if «core 1» performs a memory access to «memory bank 2» the memory access have to travel through the slower and more latent interconnect. On these architectures it makes sense to migrate memory section between memory banks, if a processor is performing a lot of accesses to a far away memory bank. In operating systems like Linux there is a page migration mechanism that migrates pages between memory banks to minimize the distance, and thus the latency, between the memory and the processing units accessing the memory [12]. NUMA is a technique currently used in larger systems with several physical chips and large memories, although the general NUMA idea with hierarchical memories can well be suited for many forms of architectures [11].

On shared memory SMP architectures the preferred method for balancing load between cores has been simply to move a task from the run queue on one core to another cores run queue. However, there are some concerns that current operating systems running on shared memory SMP systems will not scale up to 1000+ cores [13, 6]. The reason for the concern is that current operating systems have data structures that are concurrently accessed by several processing units. The most frequent synchronization mechanism between accessors is locks. To increase concurrency the trend is to add more locks, adding more overhead to the operating system code. It is only possible to lock the data structures down to the smallest data entity, after this it is not possible to increase the concurrency further. The solution is to let the operating system state become more distributed instead of shared and go towards explicit message passing instead of memory sharing [14]. The consequence is that the preferred method of balancing load introduced by tasks, goes towards task migration. On shared memory SMP systems load balancing has been relatively simple, however, due to the increasing number of cores on future chips, data structures in operating system will have to be distributed. This means that the data dependencies of a task will have to be moved, when the execution of the task is moved to another processing unit, and thus the task will have to be migrated.

In a system with a distributed memory architecture there are several separated
memory spaces, in contrast to shared memory architectures where the memory space is shared between several processing units. Distributed memory architectures can take many forms, and tends to scale better than shared memory architectures in term of number of computing nodes, this can especially be observed in the HPC field. Figure 2.2 illustrates a distributed memory system to demonstrate differences between a shared memory system, illustrated in Figure 2.1 and a distributed memory system. This system consists of two «SMP systems» with an interconnect between them. The structure of the interconnect is rather arbitrary – for example it could be an interconnect on a network-on-chip, like the connections between switches in Figure 2.6, or the «SMP systems», in Figure 2.2, could have Ethernet or Infiniband connections between them to form a cluster as in many HPC clusters. Distributed memory systems differs from shared memory systems in that they do not share all memory, and must thus rely on message passing.

When comparing distributed systems to shared memory systems, explicit message passing gives both strengths and weaknesses to distributed systems. The strength of message passing is that nodes are isolated from each other, meaning that nodes will not interfere with each other and cause congestion in systems with many nodes. The strength of shared memory is that read and writes can happen at any time from any node, leading to the weakness of message passing which is the fact that communication have to happen explicitly between nodes. Figure 2.2 is illustrates a distributed memory system. For the sake of generality the system consists of two many-core SMP systems that are connected together with a bus requiring explicit communication. If «core 1», residing on «SMP system 1», requests an access to a resource located on «SMP system 1», the operation can be performed directly inside the system, but access to a resource in «SMP system 2» must be done via message passing. Even though communication in distributed memory systems is required to happen via explicit message
passing it is possible to use virtual memory to build an operating system abstraction that hides the separation and can provide an interface with the same properties as a huge shared memory. A such interface is provided by the cluster operating system MOSIX [15]. Even though shared memory architectures with cache coherence may scale to thousands of cores [11], the operating system structure used today may not scale such an hardware [14]. In distributed memory systems explicit message passing is enforced, but in the future programming methodologies from distributed memory systems may have to be adopted even on shared memory systems, therefore it is important today to be aware of the challenges involved with programming distributed memory systems.

Heterogeneous architectures are used in many fields today to increase performance and reduce power consumption. SoCs are built with different kinds of processors for different purposes to enable performance critical applications and to maximize the idle time of the chip. These kind of special purpose processors are makes it feasible to produce battery powered chips capable of playing full-HD movies with general purpose computing capabilities. Many new GPU (Graphical Processing Unit) designs include functional units to enable execution of more generic code to open up possibilities for programmers to use the processing power earlier only used for shaders in graphical programming, this change in the GPU architecture is known as GPGPU (General Purpose Graphical Processing Unit). This tremendously increases the performance of many applications, because of the huge processing power of GPUs. GPGPUs use less energy per floating point operation, and can thus be used to reduce power consumption.
and reduce costs of the cooling of datacentres. Heterogeneous architectures are emerging because it is not longer possible to make feasible homogeneous architectures that fulfill the performance requirements of today’s applications, especially when power and energy requirements are also considered.

Examples of heterogeneous architectures are Texas Instruments OMAP series SoC’s which is widely used in mobile phones and other multimedia oriented embedded devices; and a GPGPU used together with a standard CPU is a heterogeneous architecture capable of performing one or two orders of magnitude more flops than a homogeneous standard CPU architecture would be capable of.

From the multiprocessing field of hardware architectures has been presented three different types. This thesis will cover focus mostly on homogeneous shared memory architectures, even though distributed and heterogeneous are important as research subjects. The thesis will continue with going further into detail about the hardware platform most relevant for this thesis.

2.4 Chosen platform

The hardware platform used for idea testing and development of demonstration applications is presented in this section. In the previous section three different types of hardware architectures where discussed, but mainly due to time constraints, it is not possible to evaluate all of these platforms. It is possible to later extend the scope of the implementation, discussed in Chapter 4, to cover other types of hardware architectures.

The software platform chosen for the implementation is FreeRTOS [7]. FreeRTOS is a real-time operating system capable of running on many hardware architectures. The source code of FreeRTOS is very easy to overview and has relatively simple internal structures, which makes it very easy to understand the operating systems inner workings. FreeRTOS provides a single-processor scheduler and some intra-core IPC semantics. In conclusion, FreeRTOS provides some necessary services, and stays in most situations out of the way from the programmer. This provides a suitable platform for low-level operating system functionality programming. Another reason that FreeRTOS was chosen is that Wittenstein is also involved in the RECOMP [5] project. Wittenstein provides support for the commercial variant of FreeRTOS, going under the name OpenRTOS. If the implementation had been based on the Linux kernel, possible
changes to, and the understanding of the internal structures of the kernel, would have been much more costly than on FreeRTOS. FreeRTOS is a good choice because of the low-level approach to the problem.

The hardware platform chosen for the implementation is the Versatile Express. An block diagram of the Versatile Express is presented in Figure 2.3, from the figure can be seen that a big part of the platform is built with a FPGA to provide reconfigurability of the platform. The Versatile Express is a SoC prototyping board, and can be equipped with two changeable daughter boards. During this work the Versatile Express was equipped with one CoreTile Express A9x4 daughter board. This daughter board has a Cortex-A9 quad-core cpu [2] and is equipped with 1 GB of ram. The large memory frees the development and testing process from memory constraints. FreeRTOS was ported to the platform [16] and the platform is overall well documented, which makes the platform well suited for developing near-hardware oriented software.

![Block diagram of the Versatile Express](image)

Figure 2.3: Block diagram of the Versatile Express [1]

The quad core Cortex-A9 hardware being a shared memory SMP architecture, implies that the system does not have distributed memory, and the lack of accelerators
concludes the platform not being heterogeneous. This fact directs the work towards shared memory systems instead of distributed memory systems.

### 2.5 Task migration scenarios

To facilitate a discussion of how task migration and run-time updating should be performed, eight different scenarios have been identified. In Figure 2.4 is presented an architectural diagram where the scenarios appear. In Table 2.1 the scenarios are presented in form of the capabilities required of the participating systems. The identified scenarios appear in situations where tasks are migrated in the three types of architectures discussed in the previous section, when taken into account the possible existence or non-existence of virtual memory on the source and target systems. From these scenarios it is possible to derive strategies to implement task migration between a pair of processing units. Run-time updating is identified as one of these scenarios, and is thus seen as a special case of task migration. In the rest of the section follows more in detail descriptions of these scenarios.

In scenario 1, task migration happen inside a shared memory system. Scenario 1. appear in systems with task migration between cores in shared memory systems. These systems are often SMP systems including multi-core x86 systems, the Cortex-A9 MPCore [2] and Tileras tile-processor [3]. Scenario 1. is today performed in operating systems by moving a task from the run queue of one core to another. However,
as core count on chips increase there are concerns that operating systems of today will not scale to a huge processor count [14], even though on-chip cache coherency might [11]. Scenario 1. is probably the most commonly occurring scenario presented here, and is handled by current operating systems numerous times per second. Today scenario 1. is performed essentially by doing a task switch between cores. However, the method might have to be revised in the future because of increasing on-chip core count.

Two shared memory SMP architectures scenario 1. can be performed on is presented in Figure 2.5, illustrating a block diagram of a Cortex-A9 MPCore chip, and in Figure 2.6 is presented a block diagram describing the network topology of a Tilera TilePro-like network-on-chip. These architectures have quite different layout and performance characteristics, even though they are both cache coherent shared memory SMP systems. The Cortex-A9 in Figure 2.5 is a quad core, where all the CPU cores are connected to the main memory bus using a crossbar switch. All cores have a MMU, which means that the cores have virtual memory. The Tilera TilePro-like system shown in Figure 2.6 is a 9-core network-on-chip, although real TilePros have larger core count, usually 36 or more. Here all cores are connected to a private switch which in turn is connected to a network. The network is further connected to the peripherals, such as the main memory. Memory accesses will be routed between the
switches through the network to the corresponding memory controller, in case there are several memory controllers. Every switch contains a cache, and a distributed cache coherency protocol maintains the distributed cache, transparently to the programmer. In addition, every switch contains an MMU, which means that the cores have virtual memory, similar to the x86 and the Cortex-A9. These are the two essential shared memory SMP systems, where scenario 1. could arise. In the future it is expected that the core count will rise, but from an operating system standpoint the architecture will most probably remain.

Scenario 2. presents task migration between processing units with different ISA and a shared memory. This is often the case in SoCs which have an embedded accelerator such as a GPGPU or a DSP. These types of systems has been common in mobile phones and other embedded devices for some time, but migration between different processors is not being practised. The accelerators are often optimized for high throughput, and control flow is often handled by the CPU, rendering scenario 2. not being used in practice on these types of platforms.

Task migration between processing units with different ISA and shared memory is known as scenario 3. when virtual memory is available. This type of setup could
be present in SoCs with embedded accelerators, similar to scenario 2., but with the difference of having virtual memory available. There are not many devices fulfilling the requirements for this scenario.

As an example is in Figure 2.7 illustrated the memory remapping hierarchy of the Raspberry Pi. It is unclear if the device gives rise to scenario 2. or scenario 3. The SoC differs from many others in the way that there are two levels of MMUs, as seen in Figure 2.7 [17]. The hierarchy makes it unclear which scenario the Raspberry Pi gives rise to. According to the datasheet, the lower level MMU is a coarse grained one, but the datasheet does not further state any of the MMUs capabilities or configuration possibilities. If the MMU for the GPU provides the remapping capabilities necessary for task migration the device could give rise to scenario 3.

In distributed memory environments, where all participant have the same ISA and virtual memory is available, the task migration process will conform to scenario 4. One of the most common situations for this scenario is when task migration happens between cluster nodes in a network such as Ethernet. A computer cluster has a distributed memory architecture, when classified as one of the architecture types dis-
Figure 2.7: An overview of the processors and the MMU’s in the Raspberry Pi.

cussed in previous sections. Virtual memory in the cluster nodes enables transparent migration, and there are implementations of this scenario made on existing operating systems. One example of an implementation of scenario 4. is Mosix [15], which is a mature implementation of this scenario. In Mosix, the migration is transparent and a task consists of two parts: the remote and the deputy. In Mosix only the remote is migrated and a transparent link is established between the remote and the deputy. In this scenario the only method of moving a task from one processing node to another is task migration. Scenario 4. is the traditional scenario for task migration, a scenario where moving of tasks cannot be solved by any other methodology. However, since on-chip core count go up and operating system structures change, traditional load balancing methods may not be sufficient in the future, the scenarios, in which, it will be required to perform task migration could increase in number.

The case for scenario 5. is a distributed memory environment, homogeneous in terms of ISA, but with no virtual memory. This scenario is less frequent than scenario 4., because when task migration is desired it is often possible to use systems including MMUs and thus avoid the awkward challenges of scenario 5. However, a domain
where this scenario could be proven relevant is embedded systems without virtual memory, where it is not feasible to have an MMU.

Scenario 5. differs from scenario 4. in the way that there is no virtual memory available’ which makes it difficult to migrate tasks if the memory layout of the systems are not exactly the same. The difficulty comes from the necessity to remap the memory regions of migrated tasks if the participating systems memory mappings are not the same. In this scenario, when the question is about embedded systems with possible real-time requirements, a more lightweight and less transparent execution migration solution would be better suited. A more lightweight approach would be to attempt not to migrate the tasks entire memory image, but rather to migrate some algorithmic parameters, which are specifically designed for the implementation of the application. The lightweight approach would lead to a smaller transfer size, with the downside of having a less transparent and more time consuming migration process to implement. Developing an application for this migration scenario with the methodologies in the thesis is similar to developing an application with run-time updating support. The advantage of such an implementation is that it can be used for both scenarios.

In a system with at least two ISAs participating in a distributed memory migration, scenario 6. encountered. Scenario 6. is known as heterogeneous task migration with distributed memory. The scenario usually implies reconstruction of the stack and the heap [18], because of incompatibilities between different ISAs. Existence of virtual memory is irrelevant, since if there are two differing ISAs the stack and the other memory sections will have to be reconstructed in either case. This scenario is probably the hardest to implement and most costly to perform. If the scenario is implemented as transparent as possible, it is required to have the user-state broken down to a serializable object from the source architectures ABI, transferred and reconstructed on the target system with the target systems ABI.

Even though this scenario brings big costs, people have tried to do this, even transparently [18]. This scenario has applications in datacenter environments, one could find situations where it would be meaningful to migrate tasks from powerful architectures to energy efficient architectures when the load is low to minimize resource slack, and vice versa when the load is high to achieve better performance. This scenario does also arises when migrating applications between central processing units and accelerator processing units where the memory between the two is not shared.
Load balancing between a Intel MIC expansion card and a Intel x86 CPU gives rise to this scenario, since the Intel MIC architecture does not have the standard x86 ISA. Barriers between heterogeneous architectures will probably remain hard to overcome. Even java, which has been around for about twenty years has not yet proved to entirely remove barriers between hardware architectures.

Considering the last scenario without virtual memory, the obtained scenario is scenario 7., the scenario is thus also known as homogeneous shared memory task migration without virtual memory. This kind of migration scenario arise in MPSoCs where virtual memory is not available [19].

Systems with virtual memory can be configured to behave as statically memory-mapped systems. This observation implies that systems giving raise to this scenario 1. can often be configured to behave like a system giving raise to scenario 7. if virtual memory is mapped in a special way. One example of a possible mapping that avoids assuming virtual memory is presented in Figure 4.10 and further discussed in conjunction with the task migration implementation in section 4.9.1. There are very few on-the-shelf systems which have hardware with a property that gives raise to scenario 7., not counting dynamically configurable virtual memory systems. It is possible to imagine many-core embedded systems without virtual memory, where cores shares memory. In these kind of systems it is possible to perform task migration efficiently between the cores if the tasks user-state resides in the shared memory. The interest of investigating operating systems with distributed properties leads the thesis towards this scenario. The lack of available hardware head the thesis in the direction of configuring a virtual memory system to simulate this property. When the on-chip core count goes over the scalability limit for current operating systems a solution with similar notions could be a candidate to provide scalability for future operating systems.

Using the methodologies presented in the thesis, run-time updating is identified as migration scenario 8. Run-time updating is migration in the sense that the state of a task running one version of the software is migrated to run in a task with a new version of the software. Run-time updating does not require virtual memory, and the update is performed in the same memory space. The task migration methodology discussed in the thesis is transferring a tasks state from one machine to another. Run-
time updating is therefore understood as moving a task's state from one version of the software to another. An application supporting task migration scenario 5. can easily be extended to support run-time updating and vice versa. More specifics of run-time updating is further discussed in section 3.2.

This section presents eight scenarios involving task migration and run-time updating, involving the architectures discussed in section 2.3. The scenarios present a division of the task migration problem space, and provides a way to limit the field of study. The scope of the thesis does not involve heterogeneous task migration which means that, both systems participating in a migration will have the same ISA. Accordingly, from Table 2.1 we see that scenarios 2., 3. and 6. will not be dealt with further. The used hardware platform, in the scope of the thesis, does not have distributed memory, which leads us not to study scenarios involving this kind of architecture. The implications are scenarios 4. and 5. not being further investigated. In conclusion, scenarios 1., 7. and 8. are left for further analysis. The hardware platform we use will directly give rise to scenario 1., but the complexity of implementing a dynamic virtual memory solution drags the scope of the thesis towards scenario 7. Scenario 8. is run-time updating and is included in the scope of the thesis.
This chapter describes the methodologies and declares the functionality required by
the implementation described in Chapter 4. Task migration is addressed as transferring the task state from one machine to another. Run-time updating is addressed in a
similar way, with the difference that the task state is transferred from one version of
the software to another. This way the methodologies are built on the notion of a task state. In the previous chapter, a division of the task state into two parts was discussed. These parts are the user-state and the OS-state. The previous chapter described the environment, in which task migration and run-time updating are studied in. This chapter will describe the methodology aspect of these notions, along with the concepts needed to understand the methodologies.

The task state is the most important aspect of the software part of the platform model describing the notion of a task in the context of the thesis. When activities like task migration or run-time updating are performed it is required to know that the task is in a state capable of such an activity. A state supporting a such activity, is called a checkpoint. A task must be in a checkpointable state before an activity can be performed on it. In practice a subset of task states is introduced, in which every task state supports such activities. Each state in this set is known as a checkpoint or the equivalent term checkpointable state. All other states in the entire task state space are called inconsistent states.

The problems addressed in this thesis, run-time updating and task migration, are solved by transformations of the task state. Task migration is thus performed by transforming a task state. Transformations should not be performed on the task state itself, since the task state is a object residing inside the operating system kernel. The transitive activities should be considered operating system independent and should likewise be performed on objects that conceptually are operating system independent. This requires a operating system independent representation of the task state, therefore the
notion checkpoint object is introduced. A checkpoint object is a representation of a task state being a member of the subset of checkpoints. A checkpointable state is transformed into a checkpoint object. The notion of a checkpoint object is introduced to distinguish between task states and objects that it is possible to perform transformations on.

A checkpoint object is a representation of a checkpointable state, with volatile properties. A checkpoint object should have properties that enables the operations, required by the activity, to be performed. For example, for the purpose of doing a task migration in a shared memory SMP system, the checkpointable state can be identical to the checkpoint object, because the transfer between the two machines has very high bandwidth, and it is desirable to be able to guarantee a small transformation time of the task state into a checkpoint object. Another example is distributed memory migration over internet, i.e. task migration scenario 4. Considering this scenario, we want the checkpointing object to be small in size, the long transfer time makes it beneficial to spend time minimizing the checkpointing object, and it is thus feasible to spend time waiting for the task to enter a task state yielding a smaller checkpointing object. For example if a computation with a huge amount of temporary data is executed, it is possible to wait until the computation is ready and a large amount of data can be freed and the size of the resulting checkpoint object will shrink. In the case of homogeneous task migration on a shared memory architecture i.e. scenario 1. or 7., in these scenarios it is beneficial to explicitly transfer as little state as possible. In some cases it is possible to only transfer a pointer to the state of the task. In this case the memory image of the checkpoint object could in this scenario be seen as the same as the task state, the overhead of transformation is here negligible. The observation made should be that the representation of the checkpoint object can therefore differ radically between scenarios.

Checkpointing is a notion often associated with fault tolerant systems. Fault tolerance is achieved, using the methodologies in this thesis, by saving a checkpoint object to persistent storage. In case of failure, the checkpoint object can be transformed back into a task state and the computation can continue from a previous state. The methodologies described in the thesis are capable of achieving fault tolerance, but this will not be covered in the scope of this thesis.

To support the task migration and run-time updating methodologies, a few notions are introduced. The notion of the task state is already introduced in Chapter 2 along
with the notion of the *task model*. To put a task into a state where it is possible to perform desired operations on it, e.g. migrate it or update it, we put the task into a *checkpointable state*. In such a state it is possible to *transform* the *task state* into a *checkpoint object*. If a *task* at some point in time is not in a *checkpointable state*, the *task* is in an *inconsistent state*. If an operation, such as task migration or run-time updating should be performed on the *task* the task should be executed until it reaches a *checkpointable state*. The notions introduced above will be utilized when the migration process is described.

### 3.1 Description of the migration process

After introducing the necessary notions, it is possible to describe the methodologies in better detail. The methodologies for task migration and run-time updating are first described as only one process, and is in the following sections of this chapter described more in detail. The process described will be called migration process, since it better resembles a generic migration process, than a generic updating process. The step-by-step description will here follow and the process can also be followed from Figure 3.1.

1. Execution of the task starts. This is a cold start, i.e. the task state is set by the programmer.

2. The task executes for a while. The task is in an *inconsistent state* where it is not possible to perform any actions on the *task*.

3. The task reaches a *checkpointable state*. The execution time between the *inconsistent state* in step 2. and the *checkpointable state* in step 3. is called $t_{CP}$. At this point it is possible to perform actions on the task. The set of actions that are performable at this point is depending on what type of *checkpointable state* the task is in.

   Example 1: The *task* reaches a *migratable state* at which it is possible to perform a *task migration*.

   Example 2: The *task* reaches a *run-time updatable state* at which it is possible to perform a *run-time update*.

4. The *task state* is transformed into a *checkpoint object*. The *checkpoint object* has different capabilities depending on the kind of action is performed on the *task*. 

23
5. The *checkpoint object* is transformed. This is an optional step and how it is implemented may vary.

Example: If the *task state* layout changes between versions, when doing runtime updating, the *checkpoint object* have to be transformed to match the *task state* layout of the new software version.

6. A *decheckpointer* takes a *checkpoint object* and transforms it into a *task state*.

7. The *task* continues execution. It is possible to see this as a hot start of the *task*. After a while the *task* reaches the *equivalent state* of the *inconsistent state* in step 2. The time is takes to get from step 6. to step 7., i.e. the time it takes to get from the *checkpointable state*, *decheckpointed* from the *checkpoint object*, to the desired *inconsistent state* is called $t_{dcP}$.

![Figure 3.1: Illustration of the migration process.](image_url)

### 3.2 Run-time updating methodology

Run-time updating is relevant in systems where a restart of the whole system is expensive. Run-time updating enables the possibility to safely update or reconfigure a part of the system at a lower cost. The goal is to let a executing non-critical part of the software be updated to a new version with little or no delay, while the system is running.
The granularity aimed for is updating of software modules, from one version to another while the system is running. Another possible aim could be to enable other types of run-time reconfigurability.

The version of the software can differ arbitrarily but should in some sense be compatible. In order to guarantee compatibility between versions we require that the program has been defined with a set of checkpoints, in which the task is in a known state from where it is possible to do a run-time update. Figure 3.2 presents a more detailed illustration of the run-time updating process. The figure shows the steps in the migration process versus the version of the software currently running.

The updating mechanism assumes that the software is able to change arbitrarily. This means that the stack will be rendered useless, the layout and content of the heap is tied to the old version of the software and is not generally compatible with the new software version.

To enable transformation of the tasks state from one version to another, the programmer has to define a section in every version of the software. The section is illustrated in Figure 3.3 as the «rtu data» section, in both versions of the binary. This section will be called the run-time updatable state, and will be included in the checkpoint object when run-time updating is performed. This is also the input to the programmer-supplied version transformation function. This function takes a state of a version other than its own, and outputs a state with the version of its own. The function guarantees thus compatibility between software versions. If the difference between the versions of the software is small, i.e. there is no difference in task state layout, the version transformation function does not need to perform any computation.

### 3.3 Task migration methodology

Process migration is the act of transferring a process between two machines [20]. Scenarios 1. – 7. in section 2.5 represent transferring a task between two machines. Task migration becomes more and more essential as an operating system feature, because the on-chip core-count increases and computer architectures evolve into more diverse and distributed systems. The more diverse the machines the more complicated is the process.
Figure 3.2: Illustration of the run-time updating process, viewing change of the tasks content with time. This is the process described in Figure 3.1 when applied on run-time updating.

Figure 3.3: Figure showing the data sections in a binary with a programmer defined «rtu data» section, specifying the run-time updatable state.
3.3.1 Shared memory migration

As discussed in Chapter 2 the chosen platform is a shared memory SMP system. This means that in terms of the task migration scenarios, the scenario actually implemented is scenario 1. In the section describing scenario 7, the observation was made that a system supporting scenario 1 can be configured to be a system supporting scenario 7. Task migration scenario 1 states a shared memory platform, but since a distributed operating system model is used, resources are not shared as they are in an ordinary SMP kernel such as Linux or Windows NT. In ordinary SMP kernels tasks are moved between cores by moving the task from one cores run queue to another. In shared memory task migration the task is moved from one kernel to another. The difference is that in the case of task migration the task can’t use any resources allocated on the source kernel transparently after the migration.

Since we require that a task is in a checkpointable state before a task migration can be performed there is always a latency between being in a inconsistent, non-checkpointable state and being in a checkpointable state. This latency is referred to as the checkpointing time \( t_{CP} \) and is illustrated in Figure 3.1 and represents the transition time from «task state» to «task state’» in Figure 3.4. There is also a latency from the point in time when the task has been migrated, until it can continue with its actual work. This is the decheckpointing time \( t_{dCP} \) and is also illustrated in Figure 3.1 and as the transit from «task state´» to «task state´´» in Figure 3.4.

On a SMP system we have the possibility of both having a low \( t_{CP} \) and \( t_{dCP} \). The transfer time \( t_T \), shown in Figure 3.4 will also be small since there is no need for any explicit copying. The transfer needed is handled by the cache hierarchy.

When going into a checkpoint state the task should be executing and any unmagratable OS resources should have been freed. Therefore if all resources then are migratable \( t_{CP} \) is the time elapsed to finish the current blocking activity. If all resources are migratable \( t_{dCP} \) will be 0, since the task can continue with its work instantly after migration. However, all resources are not in practice migratable, and \( t_{CP} \) will be larger than the ideal case.
Figure 3.4: Illustration of task migration scenario 1., viewing change of the tasks content with time. This is the process described in Figure 3.1 when applied on task migration scenario 1., as described in section 2.5.

3.3.2 Distributed memory migration

Distributed memory migration is in a sense more intuitive compared to shared memory migration, since when moving a task between memories migration is the only reasonable method. Distributed memory migration requires that the tasks state is explicitly transferred. It may be desirable to go into a checkpointable state that yields a checkpoint object with a smaller memory footprint. There are thus two contradicting objectives of minimizing the transfer time $t_T$ and minimizing the sum of $t_{CP}$ and $t_{dCP}$.

Earlier work has mainly focused on distributed memory migration and is often made a long time ago. In the 1980s and 1990s the main use case for task migration was to enable use of idle resources on other nodes in a cluster. For example, in an organisation several work stations could be linked together to form a cluster where task migration is performed between the nodes. Examples of operating systems with such capabilities are MOSIX [15] and The Sprite Network Operating System [21]. One kind of problem these systems tried to solve was the ability to efficiently run parallel long-lasting computations with little or no I/O. In the 1990s the only possible way to run a 12-way parallel compilation on commodity hardware, was to run the computation on twelve machines at the same time. In the 2010s it is possible to run a 12-way parallel make on a single chip, because the density of transistors on a chip has increased dramatically. This increase weakens the argument for distributed memory migration, because of the larger amount of local resources and because of the higher complexity of distributed memory migration compared to shared memory techniques.

One objective of the thesis is to focus on a changing-process of operating systems utilizing shared state to towards operating systems with more utilization of distributed
state. In contrast to much of the earlier migration work, where the objective has been to unify a distributed system. The objective makes the focus of the thesis on disconnecting task state from the operating system on the source processing unit rather than to perform the migration by starting the execution on the target processing unit in the most transparent and fastest manner possible. Because of this it is not possible to leave any partial state on the source processing unit after a migration, as is done in some implementations. There are numerous techniques in earlier work to minimize the transfer time $t_T$ [20]. In implementations with transparent migration and task-scheduling preemption, the checkpointing time $t_{CP}$ and the decheckpointing $t_{dCP}$ time is essentially zero.

Operating systems with task migration support are very rare and not widely used. However, the most widely used of these is MOSIX, capable of performing homogeneous distributed memory migration, i.e. scenario 4. MOSIX uses another migration process than the one presented in the thesis. Comparing these processes, MOSIX leaves a part of the migrated state in the source operating system, to enable transparency and to minimize the transfer time $t_T$. To discuss other distributed memory migration scenarios, there are other more academic implementations capable of heterogeneous distributed memory migration, e.g. scenario 6. Example of such systems includes the Tui system [22] and the BAG real-time distributed operating system [23]. Heterogeneous distributed memory migration is even less used than homogeneous because of the even more complex implementation.

The implementation in the thesis is based on a shared memory platform, but a objective of exploring utilization of distributed state on shared memory platforms. Therefore the implementation is made to resemble a distributed memory migration implementation, with reduced overhead of the explicit copying of state between memories. It is possible to add distributed memory migration support to the implementation, but this is not done in the scope of the thesis because of time constraints.
4 IMPLEMENTATION

The implementation described here is a part of RECOMP [5] deliverable D3.3. The methodology earlier described will be used to develop a unified run-time updating and task migration implementation.

The code for both the run-time updating and task migration scenario differ very little, therefore the implementation described here will be a unification of both scenarios using the model described in chapter 2. The program is structured so that a as big as possible portion of the code is located in reusable libraries. Demonstrations required for the RECOMP deliverables and other measurement programs are made as applications using these libraries. The scenario-specific parts of the implementation will be described at the end of this chapter in section 4.10.

4.1 Platform

The implementation is made for the Versatile Express with a CoreTile Express A9x4 daughter board. The software platform the implementation is based on is FreeRTOS [7]. The instruction for set used for ARM Cortex-A9 is the native ARM 32-bit long instruction set.

4.2 Implementation overview

The software is implemented as a set of library-like modules with as little coupling as possible. One can see from Figure 4.1 the dependencies between the modules. The small coupling is preferred since it will enable the implementation to run on a bigger variety of environments and platforms e.g. on platforms that require small footprint you can remove a lot of features that are not necessarily needed. Large parts of the library code is re-entrant even though it has not been a requirement per se.
The software modules are built upon FreeRTOS, with as few changes to the operating system as possible. The only changes to the core operating system have to do with task migration and are described in section 4.10.

4.2.1 Binaries

The implementation consists of several programs: a boot loader, the FreeRTOS kernel, the system functionality and the applications subject to run-time updating or task migration. To make the boot process more simple, all programs are packed into one single binary, which is loaded at boot time.

4.2.2 Directory hierarchy

The implementation is organized into several directories. The intention of this is to make the project structure more clear. There may be many violation to this organization in the project due to several incomplete refactorizations and a rapid development process.

App/ This directory contains code for the applications. The applications are com-
This document was created: 2nd November 2012 15:33

...piled into separate binaries that can later be loaded into the system using the task manager and the linker (section 4.4 and 4.5).

**bin/** This directory contains the targets produced by the build process.

**build/** This directory contains files created when building the targets. Most of the files are temporary files and not interesting for the user.

**gen_build_ninja.py** Python script, part of the build system, for generating build.ninja. Further described in section 4.8.

**Source/** In the Source directory resides the FreeRTOS [7] source code. During the development process an objective has been to minimize the number of changes to files in this directory, thus preserving the original code. The original code can be downloaded from http://www.freertos.com/. To run FreeRTOS on the Versatile Express a port made at Åbo Akademi has been used [16].

**System/** This directory contains the source code of the *system*. The *system* is described in sections 4.4, 4.5, 4.6, 4.7.4 and 4.7. The directory also contains configuration macros, boot-up code and other architecture specific code.

## 4.3 Code used from external sources

This section contains descriptions of code that has been imported to this project. The code imported is generic code for data structures and the Libdwarf library, used to parse debug information.

### 4.3.1 Niels Provos *<sys/tree.h> implementation from the OpenBSD project*

The OpenBSD [24] *<sys/tree.h>* [25] file provides implementations for splay trees [26] and red-black trees [27]. A user of the implementation can find the file on path `<System/tree.h>`.

**Splay trees** Splay trees are self-organizing trees that guarantees an amortized cost of $O(\log(n))$ for all standard operations [26]. The data structure is built up using splay
operations. The splay operation takes an element as an argument and brings the el-
lement down to the root while it at the same time balances the tree. The operation
changes the tree if the element given as the argument, is not already the root. Since ev-
every read and write operation causes a splay operation to happen, even a read operation
on the tree may cause a write to the memory. The data structure requires two pointers
for each node on a architecture with 32-bit pointers, such as the ARM Cortex-A9 [2],
causing an memory overhead of 8 bytes for every node.

Red-black trees  Red-black trees uses a dichromatic, i.e. red and black, scheme for
keeping the binary tree balanced. Between every operation these conditions will be
met:

1) every search path from the root to a leaf consists of the same number of black nodes
2) each red node (except for the root) has a black parent,
3) each leaf node is black.

The storage size for each red-black tree node is 3 pointers (left element, right element
and parent), as well as the colour of the node stored as a 32-bit integer. On the ARM
Cortex-A9 [2] the memory requirement is 16 bytes.

Licensing  The license of the software requires that the following notice is included
in the documentation of the redistributed software:

Copyright 2002 Niels Provos <provos@citi.umich.edu>
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
4.3.2  `<sys/queue.h>` implementation from the OpenBSD project

The OpenBSD [24] `<sys/queue.h>` [28] file provides implementation for list data structures, namely single-linked lists, double-linked lists, simple queues, tail queues and circular queues. For the user of the implementation the is available on the path `<System/queue.h>`. `tree.h` provides an abstraction for list structures which makes the code easier to read.

Licensing  The license of the software requires that the following notice is included in the documentation of the redistributed software:

```plaintext
Copyright (c) 1991, 1993
The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
```

4.3.3  Libdwarf

Libdwarf is a library for reading dwarf debugging information. Libdwarf is normally used in a posix [8] environments. When used in a FreeRTOS environment, with lacking posix capabilities, it is possible to use libdwarf’s internal I/O abstractions to handle ELF binaries directly in memory. A description of the dwarf debugging format exists in section 4.7.3.

The library is originally developed by SGI but is at the time of writing maintained by David Anderson. The software is copyrighted to SGI and licensed under LGPL. The code imported into this project is version 20111214 obtainable from http://reality.sgiweb.org/davea/dwarf.html.
4.4 Task manager

The task manager is a layer built upon FreeRTOS to handle migratable and run-time updatable ELF binaries, i.e. the applications. The main API between applications and the system is a part of the task manager. As one can see from the graph in Figure 4.1, that all the other modules in the system depends on the task manager, the reason is that the task manager provides a framework for handling tasks built upon FreeRTOS. The task manager manages a data structure for each manager task similar to the process struct existing in unix-like systems. These data structures are contained in a register called the task register.

4.4.1 Executable and Linkable Format (ELF)

The format used for binaries in this project is the ELF. The ELF is designed to be portable and can be used on a variety of hardware and software platforms. ELF is defined by the Tool Interface Standard Committee (TIS) for the Intel 32-bit architecture in [4], and the ARM extensions are defined in [29]. It should be noted that only the 32-bit ELF is used in this project, even though there is a 64-bit format.

In the implementation ELF binaries are used for two types of binaries. The application binaries and the system binaries. At boot-time the system binary is loaded and the execution starts. An overview of the life-time of the software is illustrated in Figure 4.9. The system binary is loaded to an address that could be determined at link-time, if the location of the system binary is not determined at link-time the system binary have to be compiled position independent. At run-time the system loads in and starts up applications from application binaries. The application binaries are linked to the system binary, and possibly other application binaries or libraries. This requires that the application binaries are compiled both position independent, so that they can be loaded at any address, and relocatable so they can be run-time linked to other binaries.

Figure 4.2 shows the linkers view of a ELF binary (Linking View) and the executing program layout after loading and linking (Execution View). The sections in the Linking view contains different kinds of data. In Listing 4.1 is a listing of the available sections in a system binary. This means that the code in the binary have to be position independent but does not have to be relocatable. The listing is made with readelf, which is a command for extracting information from ELF binaries. The most interesting in-
formation in this listing is the Name column and the Flg (Flag) column. The Flg column tells us what the linker should do when loading the binary. Only the sections with the allocate (A) flag set will be included in the execution view.

The following list presents how the different sections are used.

.text  This is the main code section. The program entry point is normally inside this section.

data  This section contains statically allocated and initialized data, i.e. initialized global variables.

got  This section contains the global offset table. This table contains addresses for relocatable symbols.

got.plt  This is the procedure linkage table for the global offset table. This section contains code that uses the addresses in .got. The code in .got.plt
is automatically called from user-written code segments, such as `.text`, if the code is compiled to be relocatable.

`.debug_info` All of the `.debug_` sections contain debug information. This information can be used for debugging the code and is used in the implementation to investigate what type a variable has. This project uses `libdwarf` to parse the information in these sections. These sections does not exist in the execution view.

`.debug_abbrev`  
`.debug_loc`  
`.debug_aranges`  
`.debug_ranges`  
`.debug_line`  
`.debug_macinfo`  
`.debug_str`  
`.debug_frame`  

`.shstrtab` `.shstrtab` and `.strtab` contains the names of the sections and symbols in the binary. The table is an array of null terminated strings. A string is addressed from various locations in the binary using an offset into one of these tables.

`.strtab`  

`.symtab` This section is a array of structs describing symbols in the binary. The struct tells, among other things, what name a symbol has, what section is belongs to, and depending on the type of symbol, where it is located. The name of the symbol is described as an offset into `.strtab`.

The sections in a ELF binary is utilized in various ways in different modules of the implementation. In the linking phase the symbol and string tables are used as a database for the symbols in the binary. The loader loads all sections with the A (Allocate) flag set. These section include in normal case code sections (e.g. `.text`) and the
data sections (e.g. .data, .got*, .bss). The sections with NOBITS Type does not contain any data and are all zeroed out, the most common example of a NOBITS section is the .bss section of a program.

### 4.4.2 Functionality

The most important part of the task manager is the task register. The task register is a singleton that keeps track of all of a tasks allocated memory sections. These sections include the sections allocated by the loader i.e. the sections in the ELF binary marked with the A flag, the stack and the dynamically allocated memory sections, i.e. the sections allocated with malloc()).

Every task register entry is a item in a search tree, and the task register is often referred to as the task register tree. The key used for the search tree is the name of the task. This means that when the name of the task subject to be updated or migrated is known it is possible to lookup the task register entry and perform the desired operation on it. The search tree is implemented using the red-black trees in OpenBSD’s `<sys/tree.h>` implementation presented in section 4.3.1.

Every task register entry contains a dynamic section register. It is the memory
allocator, i.e \texttt{apptask\_free()} and \texttt{apptask\_malloc()}, who is maintaining the dynamic section register. The dynamic memory section register keeps track of which dynamically allocated memory sections belongs to which task. This register uses splay trees from OpenBSD’s \texttt{<sys/tree.h>} to keep track of memory sections. The register is used during pointer tracing to figure out what pointers are pointing to valid memory sections. The pointer tracer is explained in section 4.7. When a task is freed the dynamic section register is used to free the tasks dynamic memory resources.

### 4.4.3 The application binary

There is a distinction between 	extit{system binaries} and 	extit{application binaries}. The 	extit{system binary} contains FreeRTOS and other system functionality. The 	extit{system binary} is started up by the 	extit{boot loader} and loads in 	extit{application binaries}. Only 	extit{application tasks} are able to be subject to run-time updating or migration.

The application binaries are built as shared ELF binaries with position-independent-code. This means that the binary have to be linked to function, e.g. \texttt{printf()} does not have to be included in the application binary, because it is linked to the 	extit{system binary}. Linking involves all non-static symbols, this include intra-binary symbols. In practice this means that the binary have to be linked both to itself and to the 	extit{system}. It is also possible to link symbols from one 	extit{application binary} to another 	extit{application binary} or a library. The linker sees no difference between symbols pointing to code or data, which means that both code and data can be shared.

For the dynamic memory section register to be able to keep track of the dynamically allocated memory section of the 	extit{application tasks}, the 	extit{applications} should always use \texttt{apptask\_free()} and \texttt{apptask\_malloc()} for freeing and allocating dynamic memory. Furthermore, applications having to capability to be run-time updated or migrated are required to have a \texttt{checkpoint request hook}, available as a symbol containing the address of a function. Examples of how a \texttt{checkpoint request hook} can be implemented is available in appendix A.

### 4.4.4 Methods

The 	extit{task register tree} contains task register entries. In the code the entries are called \texttt{task register conses\textsuperscript{1}}(\texttt{task_register\_cons}). In a object-oriented sense this data
This document was created: 2nd November 2012  15:33

type has a set of methods described below.

\texttt{task\_find()}  This method is used to retrieve, a \textit{task register cons} from the \textit{task register tree}, given a name as an argument.

\texttt{task\_register()}  This method registers a task to the \textit{task manager}, given name and a given ELF binary as arguments. After the \texttt{task\_register()} method has been invoked, which is modelled as a transition in the Petri net in Figure 4.3, the task is \textit{registered}.

\texttt{task\_alloc()}  This method allocates sections described in the tasks ELF binary. This operation is what is sometimes referred to elsewhere as \textit{loading} a binary. The operation is called \textit{allocating} here since the ELF binary already resides in memory. This transform can also be seen illustrated in Figure 4.2. The \texttt{task\_alloc()} method invocation is modelled as a transition in the Petri net in Figure 4.3. After the transition the task is considered \textit{allocated}.

\texttt{task\_link()}  This method invokes the linker and links a \textit{allocated} task. After the \texttt{task\_link()} method has been invoked, the task is \textit{linked}. This method is modelled the transition \texttt{task\_link()} in the Petri net in Figure 4.3.

\texttt{task\_start()}  This method starts a \textit{linked} task and inserts the task into the operating systems scheduler. After the \texttt{task\_start()} method is invoked, the task state is controlled by the operating system, i.e. \textit{os-controlled}. In the Petri net in Figure 4.3 this method is modelled as the transition \texttt{task\_start()}.

\texttt{task\_free()}  This method removes the task from the \textit{task register tree} and frees the \textit{task register cons}. The task should not be in a running state when \texttt{task\_free()} is invoked. The method is modelled as the transition \texttt{task\_free()} in the Petri net in Figure 4.3. One can see from the figure that the task have to be \textit{suspended} when this method is invoked. This method frees all the resources allocated for the task, which the \textit{task manager} knows about.

\texttt{task\_detach()}  This method is used in association with task migration. The method detaches the task from both the operating system scheduler and the \textit{task}

\footnote{The term \textit{cons} is taken from lisp jargon, describing an element in a singly-linked list. The element here is an element in a binary tree and the term is therefore not used in a entirely correct fashion. Another term that could have been used here as well is tree item.}
register tree. The task should not be in a running state when this method is invoked, this means that the task should be suspended according to the Petri net in Figure 4.3. After the method has been invoked the task register cons can be moved to another operating system instance if desired. This method is modelled as the transition task_detach() in the Petri net in Figure 4.3.

task_attach() This method attaches an earlier detached task register cons to the operating system scheduler and the task register tree. After this method is invoked the task is in a operating system controlled state. This method is modelled as the transition task_attach() in the Petri net in Figure 4.3.

task_call_crh() This method calls the tasks checkpoint request hook, if it is available in the application binary, if it is not available the function returns an error code. After this call the task should go into a checkpointable state.

task_wait_for_checkpoint() This method blocks the current task and waits until the given task register cons’s task is in a checkpointable state.

apptask_malloc() This method calls malloc() and, on success, registers the memory section in the tasks dynamic memory section register.

apptask_free() This method unregisters the memory section from the tasks dynamic memory section register and, on success, calls free() on the given memory section. The memory section is thus only freed if it was earlier correctly allocated with apptask_malloc() by the same task.

4.4.5 Task start-up process

The task manager requires a certain process to start up a task from a ELF binary. This process is a fireable sequence on the Petri net presented in Figure 4.3.

1. task_register() registers the task. The method takes two arguments, a task name and a pointer to a ELF binary.

2. task_alloc() allocates necessary memory sections to the task. In practice the linker reads the ELF binary and allocated all sections with the allocate flag set. This has the implications that the code and data sections are copied and the NOBITS sections are allocated and zeroed.
3. `task_link()` links the allocated sections. This will be further explained in section 4.5.

4. `task_start()` inserts the task into the operating system scheduler. After this, the task can be put into a running state by the operating system.

5. `task_free()` frees the task when the task should be terminated, before this the task have to be suspended by the operating system.

4.5 Linker

The linker provides an interface to some low-level ELF algorithms as well as to linker algorithms needed for run-time linking ELF binaries. This interface is mostly used by the task manager. For the use cases presented in the thesis, the user will only be interested in directly accessing the function `check_elf_magic()` from this module.

4.5.1 Methods

`check_elf_magic()` This function checks if the magic number is correct for a ELF binary. The magic number should always be checked before attempting to parse the ELF binary format.

`find_section_index()` This function finds the index of the ELF section with the given name. If no section is found the function returns 0.

`find_section()` This function uses `find_section_index()` and performs further one lookup to find a pointer to the section. If the section is not found, the function returns NULL.

`find_symbol()` This function performs a lookup on the symbol with the given name. If the symbol is found a pointer to the named symbol is returned, otherwise the function will return NULL.

`link_relocations()` This function links all dynamically relocatable symbols found in the given ELF binary [29]. The function takes a pointer to the `system ELF` and a `task register tree` pointer as arguments. These ELF binaries are used to perform lookups on missing symbols in the following order
Figure 4.3: Petri net describing the state flow of a task managed by the task manager. The Petri net shown here is in its initial state.
1. Task register tree, in-order search, by name

2. System ELF.

If one assumes that the lookup complexity for one symbol is $O(n)$, where $n$ is the total number of symbols in all given ELF binaries. If there furthermore is $m$ symbols in the binary currently being linked, the complexity of the linking algorithm will be $O(n \cdot m)$. This complexity could be reduced if a more clever data structure would be used for looking up the symbols in all the given ELF binaries. Although it should be noted that the linker does not introduce any dominant overhead for binaries with a small number of symbols.

4.6 Migrator

The role of the migrator is to control the sequential flow of a run-time updating or task migration process, illustrated by step 3. – 6. in Figure 3.1. The actors involved in this process is illustrated in Figure 4.4. The actual sequential flow is dependent on which migration scenario is implemented. The two implementations of the migrator functionality is further described in section 4.10.

4.6.1 Functionality

migrator_start()  This procedure initializes the migrator and starts a task executing the function migrator_task().

migrator_task()  This procedure is the non-returning task function for the migrator. This function is sent to FreeRTOS’s xTaskCreate() function, from the migrator_start() function. migrator_task(), in turn calls migrator_loop(), in which the user-implemented main loop resides.

migrator_runtime_update()  This procedure performs a run-time update on the given task register cons and replaces the old software with the new software given as an argument in the form of a ELF binary. Further details on what this procedure does is covered in section 4.10.1.
4.7 Pointer tracer

To extract certain aspects of the state according to the model described in Chapter 2, a pointer tracer is implemented. What the pointer tracer does on a high level is to follow memory references and mark the object they are contained in accordingly.

4.7.1 High-level description

C is used as the programming language. In the C language the only non-atomic data types are pointers, using Knuths garbage collecting terminology [30]. What the pointer tracer does is building up a memory layout graph, that can later be used to investigate which pointers point to which memory objects. The memory layout graph is built up of memory references and memory variables, and is thus a directed graph.

A memory object is a linear and continuous section of memory. A memory object is the result of a memory allocation. A memory object can be of any positive integer size counted in bytes. The smallest entity possible to mark by the pointer tracer is a memory object. If a marked memory object contains other memory objects all the contained memory object are considered marked. This property builds up a fractal of memory objects, and for this reason the term memory section can sometimes be used interchangeably with the term memory object. Below is listed the different classes of memory objects considered in the implementation.

Atomic data variables: Atomic data variables are easiest described as the leaf nodes in a tree, but more precisely described as nodes with only in-going edges in the memory layout graph. Example of an atomic data type is the data type int, in the C language. A graph containing four int's is illustrated in Figure 4.5. When
a atomic data variable is encountered by the algorithm the variable is marked, but not followed.

\[
\begin{array}{cccc}
  \text{int} & \text{int} & \text{int} & \text{int} \\
  \text{int}  *
\end{array}
\]

Figure 4.5: Illustration of memory layout graph with a pointer to an integer array.

Pointers: C pointers are the fundamental non-atomic data types. These data variables are nodes able to both in-going and out-going edges in the memory layout graph. The data variable has an out-going edge if the pointer points to a valid, allocated memory object. Figure 4.6 is an illustration of three different situations involving pointers. Below is explanations of the different situations in the figure.

1) When a pointer points to NULL the pointer is considered invalid, the variable has no out-going edges in the memory layout graph and the pointer variable is thus atomic.

2) When a pointer points to an unknown memory object, the pointer is considered invalid, the variable has no out-going edges in the memory layout graph and the variable is thus atomic.

3) When the pointer points to a valid data variable, the pointer is considered valid, the variable has a out-going edge in the memory layout graph and the pointer data variable is thus considered non-atomic.

When a non-atomic data variable is encountered by the algorithm the variable is marked and the algorithm is recursively run on that variable.

Arrays: Arrays with elements having a atomic data type are also considered atomic. Since memory objects are fractals it follows that if a pointer points to one element in an array the whole array object is considered marked.

Arrays of pointers: When an array with elements consisting of a pointer data type is encountered the algorithm is sequentially run on every element to investigate the
Structs: C structs are collections of several variables having a linear and continuous memory organization. Structs are treated similarly to arrays of pointers.

Unions: C unions are tricky since it is virtually impossible to know what the programmers intentions are [18]. One strategy to handle unions when doing pointer tracing is to be conservative and handle unions in the same fashion as structs, with the difference that each variable inside the union is on the same address. This implementation ignores unions, and unions are thus not supported.

4.7.2 Implementation details

In order to perform pointer tracing in the fashion as described in section 4.7.1, it is required having the type information for every memory location available. It is also needed to have information about all the memory objects and memory sections worked with.

In this work the type information will be extracted from debugging information. A big advantage with retrieving type information from debugging information is the fact that the resulting code will not suffer from performance overhead while executing. A disadvantage is that memory will be used to store the debug information, and the size of the executable will be larger.

Figure 4.6: Illustration of memory layout graphs with pointers pointing to valid and invalid memory objects.

elements atomicity. A situation with an pointer array is illustrated in Figure 4.7. This functionality is not implemented.
4.7.3 Dwarf

GCC gives information about the program in the dwarf [31] debugging information format. libdwarf ² is used to parse the information. The dwarf format is defined in a way that it does not favour any compiler or debugger implementation. Therefore the definition of the format is generic and, unfortunately for purposes used here, rather abstract.

The dwarf debugging information is contained in the .debug_ sections in the ELF binary, although the standard states that it does not necessarily have to be like this. The debug sections are not copied at allocation time, which means that the binary have to be available when the dwarf debugging information is used. Dwarf debugging information is used by the pointer tracer to untangle what type memory locations have. More specifically if a certain memory location is a candidate to be a non-atomic variable.

The dwarf format stores debugging information in a compact way. Large programs will have a lot of debugging information and a even more compact format would be beneficial, therefore in a more mature implementation it could be proven beneficiary to create a dedicated format for storing the type information needed. A big advantage with the dwarf debugging information format is that GCC without any extension can insert the information into any binary. Another advantage is that the GCC dwarf implementation is stable and widely used.

Format

The information in dwarf is stored in DIEs (debugging information entries). All of these entries have a tag [31]. The entries of interest for performing pointer tracing have a tag indicating a type or a variable. This holds for programs written in the C language, but for more object-oriented languages such as Ada and C++ entries with other tags have to be taken into consideration. A DIE with a variable tag should have a type attribute. This type attribute should point to a DIE with a type tag. This type DIE can further have a type attribute pointing to another type DIE. This makes it is possible to build complex types such as const int *. As an example, the structure of this type described with DIEs is shown in Figure 4.8. The tag symbols has always a DW_TAG_ prefix, and in the same manner have all the attribute symbols a DW_AT_

²libdwarf is a consumer library interface to dwarf [32]. libdwarf was originally written by Silicon Graphics beginning about 1991. The library is now maintained by David Anderson.
prefix.

The DIEs does contain different information with different formats, the format of the DIEs are described by *abbreviations*. To make the dwarf format a more compact the DIEs and the *abbreviations* are stored separately. This saves storage space, because many DIEs use the same *abbreviation*. The the DIEs are stored in the `.debug_info` section of a binary. The standard states that the storing the DIEs in this particular location is not mandatory. The *abbreviations* are, in the same manner, stored in the section `.debug_abbrev` [31].

### 4.7.4 Dwarfif

Dwarfif is the interface the *system* uses to use libdwarf. Dwarfif provides services to the migrator and the pointer tracer. Dwarfif is sets up an interface to libdwarf and provides a DIE iterating framework to easily access the type hierarchies used by the pointer tracer. Libdwarf provides a object-oriented interface to its users in the form a `Dwarf_Debug` object, Dwarfif essentially extends libdwarfs functionality and provides a better suited interface for the implementation.

**Methods**

- **dwarfif_init()** Initializes a `Dwarf_Debug` object for use with the implementation. This method have to be invoked on a `Dwarf_Debug` for dwarfif to be used.

- **dwarfif_finish()** Finishes up and frees a `Dwarf_Debug` object.

- **dwarfif_follow_attr()** Returns the DIE that the given attribute points to. This is a part of the DIE iterating framework, and it can be used to descend one step down into a type hierarchy such as the hierarchy illustrated in Figure 4.8. It is considered an error to try to follow an attribute not pointing to a new DIE.

- **dwarfif_follow_attr_until()** Uses `dwarfif_follow_attr()` to follow an attribute, and repeats until a DIE with a certain tag is found. This method can be used to e.g. find the first pointer type in a type hierarchy.

- **dwarfif_get_type_die()** Finds the type DIE of a variable DIE. This method is used to find out which type a variable has. The returned type is the root DIE
of the type tree as described in Figure 4.8.

dwarfif_die_has_typetag()  Checks if the given DIE has a type tag. This method is useful because it checks if the given DIEs tag is any of the type tags defined in dwarf.

dwarfif_find_static_var_address()  This method tries to find the address of a static variable. The method is able to this if the variable is statically allocated in a data section and the address to the variable is given in dwarf the debug information. This method is useful for the pointer tracer for following static variables.

4.7.5 Low-level description

Here follows a more low-level description of the pointer tracer implementation, in the form of the programming interface of the pointer tracer.

pt_pstate_init()  Initialize the state of a pointer tracing process.

pt_pstate_free()  Finish up and free the state of a pointer tracing process.

pt_trace_pointer()  Trace a given pointer, more detail on the algorithm is given in section 4.7.6.

pt_iterate_dies()  This method iterates over every known DIE to the given pointer tracing process. For every DIE the callback function given as an argument is called.

pt_get_included_section_pointer()  Returns a pointer to the memory section (or memory object) which the given pointer belongs to. The address of the memory section is retrieved from the dynamic memory section register located in the task_register_cons of the traced task.

4.7.6 Low-level algorithm

The algorithm used in the implementation is described in Algorithm 4.1 and Algorithm 4.2. The trace state \( s \) makes a distinction between visited variables and included memory sections. Both of these sets are implemented as red-black trees. The visited
variables set is used to search for which variables were already visited. The included memory sections set is after the trace utilized to extract the task state, as described in section 2.

The algorithm used in the implementation is described in Algorithm 4.1 and Algorithm 4.2. The trace state \( s \) is the set containing the memory objects included in the trace. This set is implemented as two red-black trees, one containing the set of visited memory sections found in the dynamic memory section register from the traced task, and the other one containing the set of explicitly visited variables. Simply put, \( s \) should been seen as the union of the visited memory sections set and the explicitly visited variables set. In practice, the visited variables set is used to search for which variables are visited and the included memory sections set is used after the trace to extract the task state, as described in section 2.

The reason for dividing \( s \) into two different sets of visited memory objects is because of the low-level nature of the C language. At the moment when a new memory section is included in the memory section set it is not possible to know which type the other variables in the memory section have. Therefore the tracing algorithm have to continue by tracing the variables with known types. The tree containing explicitly visited variables had known types at some point in the algorithm.

The algorithm described in Algorithm 4.1 and Algorithm 4.2 has three state parameters, the trace state \( s \), the current type DIE \( d \) and a pointer to the current variable \( p \). To make the reading easier three functions are declared,

the function \( r \) takes a pointer and gives the variable referenced by the pointer,

the function \( n \) takes a type DIE \( t \) and returns the DIE pointed to by \( t \), i.e. \( n \) returns the next type in \( t \)s type hierarchy and

the function \( m \) takes a pointer \( q \) as an argument and returns the pointer to the allocated memory section \( q \) points to.

d describes the type of the variable pointed to by \( p \), this implies that the type of \( r(p) \) is always known.

If one assumes that each lookup in the visited variables tree is \( O(\log n) \) and every insert into the tree is assumed to be \( O(\log n) \), where \( n \) in the total number of explicitly visited variables. When \( n \) variables are traced \( m \) memory section will be found, and it should be clear that \( n \geq m \). Every lookup and insert into the visited memory section
are $O(\log m)$. The total time will be according to these assumptions $O(n \log n)$. The algorithm will furthermore take $O(n + m)$ amount of memory.

**Algorithm 4.1 trace_pointer**

**Require:** Trace state $s$, Current variables type DIE $d$, Pointer to current variable $p$

repeat
  if $d$ is pointer type then
    internal_trace_pointer($s$, $d$, $p$);
  end if
  if $d$ is structure type then
    for $v$: all variables in structure do
      trace_pointer($s$, type DIE of $v$, pointer to $v$)
    end for
  end if
  if $d$ is base type then
    $s \leftarrow \{p\} \cup s$
  end if
  $d \leftarrow n(d)$
until $d$ is NIL

**Algorithm 4.2 internal_trace_pointer**

**Require:** Trace state $s$, Current variables type DIE $d$, Pointer to current variable $p$

mark $p$ and include $p$ in $s$

if $\neg [m(p) \in s]$ then
  $s \leftarrow \{m(p)\} \cup s$
end if

$d \leftarrow n(d)$

if $d$ is not NIL then
  $p \leftarrow r(p)$
  trace_pointer($s$, $d$, $p$)
end if
Figure 4.7: Illustration of memory layout graphs with a pointer to an array of pointers.
Figure 4.8: An illustration of how the variable \texttt{const int *var;} defined in the global scope, would be described with the dwarf format.
4.8 Build system

Because of the rather complicated build process needed to build the binaries in this project a dedicated build system has been developed. The build system consists of a ninja build file generator written in python. To build the project the following components are needed:

devkitARM  This is the build chain, based on GCC and GNU binutils. The only successfully tested release that can build the project for the Cortex-A9 is r37.

ninja  The build system driver. Any new version of ninja should be able to build the project. The version used during the development was checked out from the projects public git repository.

python  Python is needed to generate ninja build files. Version number should not matter, only python version 2.x has been tested.

m4  GNU m4 is needed to preprocess some of the linker scripts.

uboot-tools  The uboot tools package is a optional dependency and is only needed if you have to build U-Boot images. The only needed binary from this package is the mkimage binary.

git  Git is a optional dependency, but required to check out the code and for doing development

4.8.1 Ninja build system

Ninja is a build system similar to the make build system, but with the difference that the ninja build system has had simplicity as a design goal. Ninja build files are meant to be generated by a script, in contrast to make where makefiles are meant to be written by people. In this project gen_build_ninja.py is used to generate the ninja.build file.

4.8.2 Structure of the resulting binary

In this project FreeRTOS is used and the software platform, the goal of the project is to implement run-time updating and task migration. This requires organisation of the
running program capable of loading in new programs for run-time updating. For task migration multiprocessing support is required. These requirements suggests a build system capable of building FreeRTOS and linking it to some software, for loading new programs dynamic linking of ELF binaries will be used. The multiprocessing requirements suggests running several FreeRTOS kernel on all cores on the machine in parallel. To simplify the loading of several binaries on the machine all binaries are merged into one big binary.

During the development of this project two way of boot up the system has been used. The more simpler way do start the system is to compile the main program and link it to FreeRTOS. In the linking stage all of the application binaries are embedded into the main program binary. This done in the way that the application binaries are put into a application section in the main binary. This the main binary can be loaded into the machine and started from its entry point and the execution of the main program will start immediately. The other way to do it is to compile the main program and FreeRTOS into one kernel binary, and embed this binary as well as the application binaries in a loader binary. The whole loader binary can be loaded into the machine and when execution starts, the loader will first load the kernel binary from the application section in the loader binary, and then start the execution of the kernel.

In Figure 4.9 is shown the causality between «build-time», «link-time», «boot-time» and «run-time». In this build system the loader, the kernel, and the application are first built. After this the kernel and the applications are linked into executable binaries, which are linked into the loader when the loader is linked. At boot time the loader binary is loaded into the machines memory. At run-time the loader loads the kernel binary and starts the execution of it.
4.9 Running FreeRTOS on multi-core

Since FreeRTOS is designed to be a single-core operating system multi-core architectures are not fully supported. When FreeRTOS is run on the Versatile Express [16] as such the operating system will only run on one core. There has been attempts to transform FreeRTOS into a SMP operating system [33], although our intent is not to run our system on a SMP platform. We have instead used a model where we run a separate kernel on each core.

By running one kernel per core a lot of intra-kernel and inter-core synchronization is avoided and suggest a more scalable many-core operating system [14]. One of the many disadvantage with the scheme is having to rely on task migration to move tasks between cores. A operating system relying on task migration to move tasks between processing units have the inherit advantage of being more easily ported to architectures with more spare memory models. The task model forces the tasks to be more volatile in the sense that they should not be tied to a specific kernel to be scalable, as they are in traditional systems.

4.9.1 SMP Multicore with separate kernels on each core

When running one operating system kernel per cpu core it have to be insured that the kernels won’t interfere with each other, at the same time as they should be able to communicate. A scheme was developed which is capable of performing this isolation with FreeRTOS running as operating system on the Cortex-A9 MPCore. The scheme which also requires a MMU is presented in Figure 4.10.

In the memory layout in Figure 4.10 all of the kernels on the quad core chip have a private area for the statically allocated memory ($K_n$). On each core ($C_n$) the kernels sees it’s own statically allocated memory on the same address. Every kernel has a globally visible memory section ($GVM_n$). This section is visible to every kernel and resides on separate virtual addresses for each kernel. When kernel $n$ allocates memory the kernels allocator will give an address from memory section $GVM_n$. The $GVM_n$ section is essentially the heap for kernel $n$. To have a consistent way to do inter-core communication there is also a inter-core communication section ($ICC$). This section is shared between all cores and it only used for communication.
4.9.2 Boot process

When running FreeRTOS in the multi-core mode the boot process goes as following:

1. U-Boot loads in the full binary.

2. U-Boots sets the instruction pointer of the master core (core 0) to the entry point of the loader program.

3. The loader program reads the system ELF binary and copies out the ELF sections that are marked with the A flag, e.g. normally .text, .data and .got* sections, to all of the Kn sections, as in Figure 4.10. The loader program also sets up configuration information for the cores.

4. The loader sends a software interrupt to all cores so that they will jump to their respective kernel.
5. All cores are now running, asynchronously and independently of each other. In other words, this step and forward runs on four cores on the quad core. In this step the virtual memory is set up using the configuration information stored in step 3. After this step is completed by all cores we will have the virtual layout described in Figure 4.10.

**GVM Allocator** When allocating memory in the GVM section the standard `malloc()` function is used. The allocator is configured to use `GVMn` when we are running on core `n`. In this sense is the GVM allocator a core local memory allocator. When memory is freed `free()` must check which `GVM` section the given pointer belongs to. If the pointer belongs to `GVMm`, when current core `n ≠ m` the implementation sends a message to the kernel on core `m` to ask the allocator to free the specified memory section.

### 4.10 Use-cases

In this section is described how the unified implementation relates to the addressed key problems of the thesis. The problem specific code is described in this section together with example usage of the libraries described earlier in this chapter.

#### 4.10.1 Run-time updating

The run-time updating use-case is achieved using the following modules:

- task manager
- linker
- migrator
- dwarfif
- pointer tracer

The demos for run-time updating use-case, run only on one core on the Cortex-A9 quad core, although there is no circumstance preventing the demos from running multi-core. The reason the run-time updating demos are run on single-core is the fact that the multi-core implementation has bigger memory footprint.
Run-time updating process

Run-time updating can be described as migration scenario 8. from Chapter 2. The origin task will be referred to as the task being subject to be updated. The updated task is a task that is about to be, or has already been updated. The implemented process for run-time updating is described in the following list. It is intended to implement the migration process as illustrated in Figure 3.1.

1. The task is now in a inconsistent state.

2. Wait until the origin task reaches a valid checkpoint, i.e. a run-time updatable checkpoint.

3. Suspend the origin task.

4. Allocate the updated task, e.g. fire the task_alloc() transition to put a token into the allocated place in the Petri net illustrated in Figure 4.3.

5. Link the updated task, e.g. fire a transition to put a token into the linked place in the Petri net in Figure 4.3.

6. Find the checkpoint request hook in the updated task, and store a pointer to the hook-function it in the task register cons of the updated task.

7. Transfer the run-time updatable state. The transfer is done in the following way:
   1. First find the .rtu_data sections in both the origin task and the updated task.
   2. It is considered an error is the sizes of the .rtu_data sections doesn’t match.
   3. Perform a pointer trace in every variable in the .rtu_data section of the origin task.
   4. Allocate and copy the memory sections included in the pointer trace to the updated task. In terms if the low-level algorithm in section 4.7.6 the memory ranges copied are the ranges including all memory objects contained in the trace state set s.
   5. Update all non-atomic pointer variables to point to the correct memory places in the environment of the updated task.
8. Register the updated task to the task manager.

9. Start the task, i.e. put the task into a os-controlled state.

10. Start the execution of the updated task. At this point it is up to the programmer to run a version transformation function to transform the state from some old version to the current. (optional)

11. Execute the program until it reaches the inconsistent state the origin task was in, in step 1. (optional)

In appendix A.1 is found an example of how a run-time updatable program with support for version transformation functions could be implemented.

### 4.10.2 Task migration

The task migration use-case is achieved using the following modules:

- task manager
- linker
- migrator

The implementation of task migration supports scenario 1. as described in Chapter 2, and requires the memory model described by Figure 4.10. The origin task will be referred to as the task which is subject to be migrated and the target task is the newly created task that will contain the transferred state of the origin task on the target.

**Migration process**

The process implemented for task migration is described in the following list. The intention of the list is intended to resemble the migration process illustrated in Figure 3.1. The beginning of the process is essentially the same as in the run-time updating case, but the checkpoint reached in the task migration process is conceptually not the same compared to the checkpoint reached in the run-time updating process. The reason for this is that the requirements for a run-time updatable state are different than the requirements of a migratable state. However, the code requesting and reaching the checkpoint looks the same in the example in section A.2.
The task is now in a **inconsistent state**.

2. Wait until the *origin task* reaches a valid checkpoint, i.e. a checkpoint that allows task migration.

3. Suspend the *origin task*.

4. *Detach* the *origin task*. This action will fire the task into a *unregistered* state as illustrated by the Petri net in Figure 4.3.

5. Transfer the *task register cons* to the *target*. In scenario 1. the *target* is usually another core on the same SMP system, which means that shared memory can be used to transfer a minimal amount of data between kernels.

6. *Attach* the *target task*. After this the task will be in a *os-controlled*-state according to the Petri net in Figure 4.3.

7. Execute the *target task* until it reaches the *inconsistent state* that the *origin task* was in, in step 1. (optional)

**Implementation details**

The task migration implementation takes advantage of the shared memory architecture and is this way able to minimize the amount of explicit message passing and copying of memory. The only data needed to be passed between kernels is a 32-bit pointer pointing to the tasks *task register cons*. For real-time purposes one should take into account the transfer of data between caches. No cache prefetching is done which means that when the task starts on the new core the cache is cold.

To perform the *task_detach()* and *task_attach()* operations on FreeRTOS, additional functionality is required. The source code for the added functionality resides in the file *Source/tasks.c*, in the FreeRTOS distribution. The change is the addition of two functions with the prototypes:

```c
signed portBASE_TYPE xTaskAttach( xTaskHandle pxTaskToAttach );
signed portBASE_TYPE xTaskDetach( xTaskHandle pxTaskToDetach );
```

The functions have essentially the same functionality as *xTaskCreate()* and *vTaskDelete()* , with the difference that *xTaskAttach()* and *xTaskDelete()*
does not free or allocate any data structures, instead the functions use existing task control blocks. \texttt{xTaskDetach()} removes the tasks control block from the scheduler. \texttt{xTaskAttach()} is the inverse operation, which is to insert a task control block into the scheduler.

One should note that \texttt{xTaskDetach()} does not take into consideration if the task has any blocking synchronization mechanisms. This implies that $t_{CP}$, the checkpointing time discussed in section 3.3 and illustrated in Figure 3.1 and Figure 3.4, will be $> 0$ if the task is blocked waiting for events from any synchronization mechanism.

In appendix A.2 there is an example of a application that can be migrated between kernels, using this scheme.
5 Future work and conclusions

5.1 Conclusions

The implementation of run-time updating is used to increase flexibility of mixed-critical embedded applications. The implementation has the capability to use pointer tracing to update tasks with complex run-time updatable states. If complex states are not needed the implementation does not have to use the pointer tracer, and uses less resources.

Since the on-chip core count is increasing, the scalability of state-of-the-art operating systems could be compromised [14]. A bottleneck in current operating systems is the fact that data structures in the kernel often are shared between cores. The bottleneck is often circumvented by introducing fine-grained locks. This circumvention is convenient from an evolutionary point, since the operating system can evolve from a coarse-granularly locked kernel into a more fine-granular one. The disadvantages are that locks cannot be added ad infinitum, and furthermore, locks require expensive operations to be acquired and released. The huge amounts of locks makes the code hard to read and understand, which makes the learning curve for kernel programmers higher. Current operating system evolution is, from a core-count scalability standpoint, moving in a unsustainable direction. This could call for a radical change in operating system structure. One plausible direction for this change is towards a more distributed structure.

Task migration requires complicated procedures, adds implementations complexity and run-time overhead, compared to current methods of on-chip load balancing. Operating system evolution have, for obvious reasons, always used the least complicated but still feasible solution. One advantage that task migration has is permitting isolation between processing units, while at the same time allowing load balancing. This is a desired property for an operating system changing towards a more distributed structure. An operating system with a task model that fundamentally supports task
migration will consequently open for a more distributed structure. It could be feasible to implement such a task model in the future, but a new task model would require the use of new tools, and new programming practices, which is not usually appreciated by a potentially conservative community.

The increasing on-chip core-count requires constant innovation in hardware design. Some people claim that cache coherency will not be feasible in future many-core chips, while others claim that it will be [11]. Regardless if cache coherency is feasible on future chips or not, operating system structure will have to keep up with the increasing core-count. Task migration as an instrument for load balancing can deal with both cache coherent shared memory architectures and distributed memory architectures. An operating system that performs load balancing with task migration opens up the possibility for a data centre wide operating system, which would have more volatile loads and could open for better energy-scalability than current operating systems. A operating system which fundamentally utilizes task migration could have many advantages, from on-chip core-count scalability to data centre wide energy scalability.

### 5.2 Future work

For the run-time updating mechanism implemented in the scope of this thesis, a few demonstration applications are presented to show the functionality of the implementation. These demonstration applications are not considered real-world examples and to show the full potential of the run-time updating mechanism a more safety related application, linked to a real-world example should be demonstrated.

The task migration mechanism is only implemented to support shared memory migration. Though shared memory migration is a completely valid mechanism, the full potential of the implementation could be uncovered if the task migration mechanism would be combined with the task managers dynamic memory section register to perform distributed memory migration, i.e. migration scenario 4. One possible way to demonstrate this mechanism is to use load balancing to achieve better energy scalability in a cluster by moving away tasks from nodes with low load and turn them off.

The libraries made for the task migration and run-time updating mechanisms in the scope of this thesis, have more potential than previously shown, but the lack of time has limited the demonstrated potential.

As already stated in the conclusions there could be the need for a new task model
for many-core operating systems. There has been a lot of work done in the field of operating systems research, and the field of many-core operating systems research is emerging [14, 13, 34]. This problem area would involve investigating what this new task model would be, scheduling mechanisms, migration strategies and capabilities and how to deal with heterogeneity.
BIBLIOGRAPHY


[16] Dag Ågren. Arm cortex-a9 versatileexpress quad-core freertos port. URL: http://interactive.freertos.org/attachments/token/dpijlufgs02o4vv/?name=FreeRTOSv7.1.0_Versatile_Express_Cortex_A9MPCore.tar.gz.


This document was created: 2nd November 2012 15:33


SAMMANFATTNING

NÅGON RUBRIK

Introduktion

Asd
A APPENDIX

A.1 Run-time updating pseudo-example

This section presents an example for doing a run-time updatable application using the implementation described in Chapter 4 in pseudo-C. The example describes a generic while(1)-loop program able to be run-time updated. Run-time updatable essentially means that the example does support hot-start with several versions of state, i.e. the example supports to transform a state from one version of the software into a state supported by another version of the software.

When the execution of the program starts, setup_rtu_state() initializes the state. If the program was running earlier, i.e. the program was run-time updated the state will be set to some value ≠ NULL. In this case the state is used, which is transformed to another version or used as is. When the program is subject to be updated the migrator sends a checkpoint request to the program by calling the checkpoint request hook cpRequestHook(). When a run-time updatable or migratable program receives a checkpoint request, it should put itself into requested state. This particular application supports only to be put into a run-time updatable state. After the program receives the checkpoint request it completes the program loop code once and then tells the migrator that it is safe to do a run-time update of the program.

```c
#include <App/rtu.h>

typedef unsigned int version_t;

typedef struct rtu_state_v1_t {
    version_t version;
    /*
    * Version 1 specific data.
    */
} rtu_state_v1;
```
typedef struct rtu_state_v2_t {
    version_t version;
    /*
     * Version 2 specific data.
     */
} rtu_state_v2;

volatile int rtu_requested = 0;

void cpRequestHook(cp_req_t req_type)
{
    if (req_type == cp_req_rtu) {
        rtu_requested = 1;
    }
    return;
}

static void setup_rtu_state()
{
    if (state == NULL) {
        /*
         * No previous rtu state, cold start of application.
         */
        state = apptask_malloc(sizeof(rtu_state_v1));
        if (state == NULL) {
            /*
             * ERROR
             */
        }
        state->version = rtu_v1;
        init_rest_of_state(state);
    }
    switch (state->version) {
    case rtu_v1:
        /*
         * We have the correct version, just hot start.
         */
        break;
    case rtu_v2:
        /*
         * Last time the state was manipulated by version 2 of
         * the software. We have to do a transformation of the
         * state from version 2 to version 1.
         */
        rtu_state_v1 *old_state = state;
        rtu_state_v2 *new_state = apptask_malloc(sizeof(rtu_state_v2));
        if (new_state == NULL) {
            /*
             *
A.2 Task migration pseudo-example

This section presents an example for doing a migratable application using the implementation described in Chapter 4 in pseudo-C. The example describes a generic while (1) -loop which includes a sequence of three jobs. Between each job the task
This document was created: 2nd November 2012 15:33

going into a checkpointable state.

When the execution of the program starts init_application() initializes the application. After the initialization the execution goes into the while(1) loop and starts execution of job1(). After the job is completed the task goes into a checkpointable state. In the checkpointable state the function tmSafeState() decides if the task should go into a safe state to block and wait for migration. At this point the task can safely be migrated. If the task is not migrated the execution continues with job2() and so on.

When the migrator decides that the task should be migrated it will send a request to the task by calling the function cpRequestHook(). This will make the task go into a safe state when the task is in a checkpointable state and will thus become migratable.

```
#include <App/rtu.h>

int tm_requested = 0;

void cpRequestHook(cp_req_t req_type)
{
    if (req_type == cp_req_tm) {
        tm_requested = 1;
    }
}

void tmSafeState()
{
    if (tm_requested) {
        /*
         * Go into safe state.
         */
        TASK_IN_SAFE_STATE();
        /*
         * At this point the task have been migrated, and we
         * can set up our resources and continue to do our
         * work.
         */
        tm_requested = 0;
    }
}

int main()
{
    init_application();

    while (1) {
        /*
         */
```
* Program loop part I
*/

job1();
tmSafeState();

/*
 * Program loop part II
 */

job2();
tmSafeState();

/*
 * Program loop part III
 */

job3();
tmSafeState();
}