Diss. ETH No. 21182

Compiler-Assisted Thread Abstractions for Resource-Constrained Systems

DISSERTATION
submitted to
ETH ZURICH
for the degree of
DOCTOR OF SCIENCES
(Dr. sc. ETH Zürich)

by

ALEXANDER BERNAUER
Diplom-Informatiker, University of Ulm, Germany
born September 11, 1980
citizen of Germany

accepted on the recommendation of
Prof. Dr. Friedemann Mattern, examiner
Prof. Dr. Kay Römer, co-examiner
Prof. Dr. Thomas Gross, co-examiner

2013

Für **Liesa Bernauer**, meine Treue, meine Freiheit, mein Anker,
die eine, die mich bindet — Mutter meines Sohnes.
Ich liebe dich.

Für **Marco Baur**, mein treuer Freund, mein Mentor, mein Schüler,
der eine, der wie ich — Wanderer zwischen Welten.
Ich werde dich nie vergessen.

Danksagung

Diese Arbeit wäre ohne die Hilfe vieler Mitmenschen nicht möglich gewesen. So danke ich Prof. Dr. Friedemann Mattern für sein Vertrauen in mich und für die Möglichkeit, an einer der führenden technischen Universitäten zu promovieren. Ausserdem danke ich ihm für die Freiheit, mein Thema selber zu suchen, und für die Unterstützung, diese themenübergreifende Arbeit zum Abschluss zu bringen. Ohne dich hätte sich meinen Traum zu promovieren wahrscheinlich nie erfüllt.

Ich danke Prof. Dr. Kay Römer für die wertvolle inhaltliche Zusammenarbeit und das Vermitteln seiner Erfahrung, für seine Flexibilität, seine Offenheit und für seine Bereitschaft, mich über seine persönlichen Interessen hinaus zu unterstüzten. Ohne dich wäre ich in der Isolation versunken und hätte sicherlich noch nicht — und vielleicht niemals — abgeschlossen.

Ich danke Prof. Dr. Thomas Gross für seine Zeit, mein Thema im Vorfeld inhaltlich zu begleiten, sich gründlich mit dieser Arbeit auseinanderzusetzen und schliesslich als wichtiger Gutachter für meine Dissertation zu aggieren. Ihre Unterstützung war eine Vorraussetzung für den Abschluss dieser Arbeit, die thematisch in Ihren Fachgebiet angesiedelt ist.

Ich danke Liesa Bernauer dafür, dass sie mich von Anfang an ermutigt hat, meinen Traum zu verfolgen und an mich zu glauben, dass sie mich andauernd weiter angetrieben, hat auf mein Ziel hinzuarbeiten, und dass sie mir den Rücken frei gehalten hat, damit ich diese Arbeit abschliessen kann — und all das mit grossen persönlichem Verzicht und enormem Aufwand. Ohne dich hätte ich es nicht gewagt und schon gar nicht vollbracht.

Ich danke Fritz und Jutta Bernauer dafür, dass sich mich im Geiste der Liebe, des Selbstbewusstseins und der persönlichen Entfaltung grossgezogen haben. Euer bedinungloser Rückhalt hat mich den Mut aufbringen lassen, dieses Thema trotz Zweifel und Kritik bis zum Ende zu verfolgen.

Ich danke Dr. Christian Flörkemeier für die inhaltliche und persönliche Betreuung am Ende meiner Dissertation. Ohne deine Erfahrung, deine Motivierungen und deinen Druck hätte ich diese Arbeit nicht zeitgerecht zu Ende gebracht.

Ich danke Matthias Kovatsch für seine Freundschaft, sein offenes Ohr, den Zusammenhalt, seine Bereitschaft mir immer wieder zu helfen und den ganzen Spass, den wir zusammen hatten. Es war eine schöne Zeit mit dir und ich hoffe, es kommen noch weitere.

Ich danke Markus Weiss für seine Freundschaft, seine Treue und seine persönliche Unterstüztung, für die Skiausflüge, Feiern und Grillabende. Ich hab von dir gelernt und du hast mich verändert.

Ich danke Simon Meier für seine fachliche Unterstüztung, seine Ideen und seine Bereitschaft, einem Anfänger ohne Eigennutz zu helfen. Ohne dich hätte ich kaum die fachliche Kompetenz aufbauen können, die für die Umsetzung dieser Arbeit nötig ist.

Ich danke Steffanie Cantu für ihre professionelle sprachliche Korrektur dieser Arbeit. Dank deiner Hilfe ist es nicht ganz so offensichtlich, dass ich wenig Talent für Sprachliches habe.

Darüberhinaus haben mir noch viele Menschen geholfen, indem sie mit mir über diese Arbeit diskutiert, Fragen beantwortet, auf Probleme hingewiesen und/oder Ideen beigesteuert haben. In alphabetischer Reihenfolge waren das: Andreas Bernauer, Anwar Hithnawi, Luca Mottola, Michael Pradel, Michael Schwarz, Moritz Strübe, Rico Schiekel, Silvia Santini, Simon Mayer, Wilhelm Kleiminger und Yunyan Ma.

Ausserdem geht mein Dank an alle Beteiligten der ETH Zürich für die unglaubliche Infrastruktur und die umfassenden und freundlichen Dienstleistungen.

Zuletzt möchte ich mich bei den Beteiligten der folgenden Projekte für die finanzielle Unterstützung und die Zusammenarbeit bedanken:

- das National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS); unterstützt durch den Schweizer Nationalfonds zur Förderung der Wissenschaflichen Forschung unter der Kennung 5005-67322

- das Cooperating Objects Network of Excellence (CONET); gefördert durch die Europäische Komission im Rahmen des Seventh Framework Programme unter der Kennung FP7-2007-2-224053

Abstract

Although increase of hardware resources is the general trend in information technology, resource-constrained systems continue to be of relevance. In wireless sensor networks (WSN), for example, a typical device is equipped with a few kilobytes of RAM and provides only limited computing power. Such systems aim to provide ad-hoc and long-term monitoring of physical or environmental conditions over large spatial regions. Maximizing the number of participating devices and extending the overall network lifetime is crucial for such applications. Moore's Law is therefore exploited towards lower device costs and lower power consumption at almost constant resource capabilities per device.

This scarcity of resources has various implications on how resource-constrained systems such as wireless sensor networks are designed and implemented. Besides requiring highly optimized network protocols, one particularly important consequence is the prevalent adoption of the event-based programming paradigm. This paradigm is considered well-suited for traditional WSN applications, because it can be implemented efficiently and matches the reactive nature of such applications. However, requirements of WSN applications are continuously increasing, causing higher software complexity to meet growing expectations. As a result, today's sensor networks often run IP stacks, HTTP and CoAP services, middleware, and other extensive software components. Implementing the complex control and data flows of such applications and services in an event-based manner is error-prone and the resulting code is hard to maintain. This leads to software faults that are costly and time consuming to track down, both during development and testing, and particularly during deployment and operation. Additionally, the presence of software faults entails security issues for a deployed system. To counter these problems, better programming abstractions for resource-constrained systems seem necessary.

Threads are known for overcoming many problems of events by supporting sequential control flows via synchronous functions, but existing solutions either provide incomplete thread semantics or introduce a significant resource overhead. This reflects the common believe that expressiveness has to be traded for efficiency and vice versa. With our work, we show that this trade-off is not inherent to resource-constrained systems.

We follow the approach of compiler-assisted thread abstractions, where full-fledged thread-based C code is compiled to equivalent event-based C code that runs atop an event-based operating system. This approach is promising for two reasons. First, the event-based run-time system avoids multiple preallocated stacks and the overhead of a task scheduler. And second, a compiler can, in contrast to a run-time-based thread library, perform static program analysis and thus apply application-specific optimizations.

Our thesis is that *a comprehensive thread abstraction is possible also for resource-constrained systems. A compiler, which translates thread-based applications into event-based programs, is able to combine the efficiency of event-based programming with the comfort of thread-based programming. This addresses the increasing requirements of resource-constrained systems.* We develop our thesis in the context of wireless sensor networks, assuming that resource scarcity will continue to be relevant. Existing threading solutions either provide only limited thread semantics or require more resources than common WSN devices offer. We therefore propose a compiler-assisted approach that can provide a comprehensive and efficient thread abstraction.

We support our thesis by presenting a comprehensive compiler-assisted thread abstraction concept for resource-constrained systems. To achieve this, we developed a platform-agnostic code transformation from thread-based C code to equivalent event-based C code. This enables developers to write thread-based applications without the additional costs of a thread library. Moreover, we present a way to reverse this transformation at run-time for fault diagnostics purposes. Hiding the details of the transformation and the underlying run-time system during all phases of development completes the abstraction. In order to evaluate our approach, we designed and implemented a compiler and a debugger prototype to conduct a set of experiments. Our results show that compiler-assisted thread abstractions not only outperform thread libraries, but are even almost as efficient as hand-written event-based code. The identified overhead accounts for only 1% higher RAM usage, 2% additional processor cycles, and 3% larger binaries on average, which we consider a reasonable trade-off for the gained comfort of a comprehensive thread abstraction. Additionally, the experiments demonstrate that sustaining the abstraction level for fault diagnostics is possible.

Zusammenfassung

Obwohl der allgemeine Trend in der Informationstechnik in Richtung immer leistungsfähigerer Hardware geht, bleiben ressourcenbeschränkte Systeme weiterhin von Relevanz. Drahtlose Sensornetze, zum Beispiel, verwenden typischerweise Rechnerknoten, die mit wenigen Megahertz getaktet werden und nur über einige Kilobyte Speicher verfügen. Der Grund dafür ist, dass solche Systeme eine infrastrukturlose, langfristige und geographisch ausgedehnte Überwachung von physikalischen Phänomenen ermöglichen sollen. Da hierfür eine Maximierung der Anzahl der Knoten und der Lebensdauer des Systems entscheidend ist, wird der durch das Mooresche Gesetz induzierte Effizienzgewinn für niedrigere Gerätekosten und niedrigeren Energieverbrauch bei annähernd gleichbleibenden Hardwareressourcen pro Gerät genutzt.

Die Ressourcenknappheit hat einen starken Einfluss auf Entwurf und Implementierung solcher Systeme, so dass es zum Beispiel für Sensornetze hoch optimierter Kommunikationsprotokolle bedarf. Ausserdem ist in dieser Domäne das ereignisbasierte Programmiermodell weit verbreitet, da sich dieses Paradigma effizient umsetzen lässt und es gut zur reaktiven Natur traditioneller Sensornetzanwendungen passt. Die Anforderungen an solche Anwendungen nehmen allerdings kontinuierlich zu, was zu einer erhöhten Komplexität der Programme führt. Um den steigenden Erwartungen Rechnung zu tragen, werden auf heutigen Sensorknoten u.a. IP-Protokollstapel, HTTP- und CoAP-Dienste, Middleware und weitere umfangreiche Softwarekomponenten betrieben. Die ereignisbasierte Umsetzung von Kontroll- und Datenflüssen solcher Anwendungen ist allerdings fehleranfällig, und die Pflege entsprechender Software gestaltet sich als schwierig. In der Konsequenz führt dies zu Softwarefehlern, deren Analyse und Behebung während Entwicklung und Erprobung, aber vor allem während Inbetriebnahme und Betrieb kostspielig sind. Darüber hinaus können solche Softwarefehler auch sicherheitsrelevante Schwachstellen sein. Um all dem entgegenzuwirken, sind höhere Programmierabstraktionen für ressourcenbeschränkte Systeme nötig.

Threads können durch sequenziellen Kontrollfluss und synchrone Funktionen viele Probleme der ereignisbasierten Programmierung beheben. Bei bereits existierenden Lösungen ist allerdings entweder die unterstützte Threadsemantik eingeschränkt oder es werden leistungsfähigere Systemkomponenten benötigt als solche, die typischerweise in drahtlosen Sensornetzen zum Einsatz kommen. Das überrascht nicht, da es der allgemeinen Meinung entspricht, dass es einen Zielkonflikt zwischen der Vollständigkeit einer Threadsemantik einerseits und der Effizienz ihrer Implementierung andererseits gibt. Unsere Arbeit zeigt allerdings, dass dieser Zielkonflikt keine inhärente Eigenschaft von ressourcenbeschränkten Systemen ist.

Dabei verfolgen wir den Ansatz der übersetzergestützten Threadabstraktion. Ein Compiler wandelt hierzu ein vollwertiges threadbasiertes C-Programm in ein äquivalentes ereignisbasiertes C-Programm um, welches von einem ereignisbasierten Betriebssystem ausgeführt wird. Dieser Ansatz ist aus zwei Gründen vielversprechend: Erstens ist das übersetzte Programm ereignisbasiert, so dass sowohl vorbelegte Stacks als auch der zusätzliche Aufwand eines Thread-Scheduler vermieden werden. Und zweitens kann ein

Compiler, im Gegensatz zu einer laufzeitbasierten Threadbibliothek, statische Programmanalysen durchführen und somit anwendungsspezifische Optimierungen vornehmen.

Unsere These ist daher, *dass eine umfassende Threadabstraktion auch auf ressourcenbeschränkten Systemen möglich ist. Ein Compiler, der threadbasierte Programme in ereignisbasierte Programme umwandelt, ist in der Lage, die Effizienz von ereignisbasierter Programmierung und den Komfort threadbasierter Programmierung zu vereinen. Insgesamt kann so den wachsenden Anforderungen an ressourcenbeschränkten Systemen Rechnung getragen werden.* Wir entwickeln unsere These im Kontext von drahtlosen Sensornetzen, wobei wir davon ausgehen, dass Ressourcenknappheit auch in Zukunft relevant sein wird. Existierende Threadlösungen aus dieser Domäne benötigen entweder mehr Ressourcen als ein typischer Rechnerknoten zur Verfügung stellt oder sie unterstützen nur eine eingeschränkte Threadsemantik. Wir schlagen daher vor, mit Hilfe des übersetzergestützten Ansatzes eine umfangreiche und dennoch effiziente Threadabstraktion zu verwirklichen.

Wir untermauern unsere These dadurch, dass wir ein Konzept für eine übersetzergestützte Threadabstraktion für ressourcenbeschränkte Systeme angeben. Zu diesem Zweck haben wir eine plattformunabhängige Transformation von threadbasierten C-Programmen zu äquivalenten ereignisbasierten C-Programmen entwickelt. Entwickler können damit threadbasierte Anwendungen ohne die zusätzlichen Kosten einer Threadbibliothek realisieren. Ausserdem zeigen wir auf, wie man zum Zwecke der Fehlersuche die Transformation zur Laufzeit rückgängig machen kann. Indem wir damit die Details der Transformation und der darunterliegenden Laufzeitumgebung während allen Entwicklungsphasen verbergen, vervollständigen wir die Abstraktion. Zur Bewertung dieses Ansatzes haben wir einen Prototypen eines Übersetzers und eines Debuggers entwickelt und damit eine Reihe von Experimenten durchgeführt. Unsere Messungen haben ergeben, dass übersetzergestützte Threadabstraktionen hinsichtlich der Effizienz nicht nur Threadbibliotheken übertreffen, sondern auch fast so effizient wie handgeschriebene ereignisbasierte Programme sind. Der gemessene durchschnittliche Mehrverbrauch an Ressourcen beträgt 1% RAM sowie 2% Prozessorzyklen und resultiert in 3% grösseren Binärprogrammen. Unserer Meinung nach sind diese geringen Mehrkosten aufgrund des gewonnenen qualitativen Mehrwerts einer umfangreichen Threadabstraktion gerechtfertigt. Ausserdem zeigen die Experimente, dass es möglich ist, die Abstraktion auch während der Fehlersuche zur Laufzeit aufrechtzuerhalten.

Contents

Preface

Before we start with the introduction to our thesis, there are a few important remarks about this document.

1. This document uses a consistent formatting to convey meta information of text:
   - *phrase*: This indicates a phrase that is included in the Index, the List of Figures, the List of Tables, or the List of Abbreviations of this document.
   - `name`: This indicates the name of a keyword, function, a variable, or any other identifier from a program written in any programming language.
   - **`syntax`**: This indicates concrete syntax used in the specification of code patterns.
   - `term`: This indicates a meta variable used in the specification of code patterns.

   An example would be:

   conditional assignment: **`if (`**`condition`**`)`** `var1 = value`**`;`**

2. This document references sections and clauses of the C language standard ISO/IEC 9899:1999 [64]. Such references use the following syntax:
   - "C99" refers to ISO/IEC 9899:1999 as a whole.
   - [C99: 1.2.3] refers to Section 1.2.3 of C99.
   - [C99: 1.2-3] refers to clause 3 of [C99: 1.2].
   - [C99: 1.2-3–5] refers to the range of clauses from [C99: 1.2-3] up to and including [C99: 1.2-5].

3. This document also contains references to sections of the GCC manual [127]. The *GNU Compiler Collection* (GCC)[1] is mainly an implementation of the C programming language family. The employed references are used to refer to the various GCC language extensions. Such references use the following syntax:
   - "GNU C" refers to the GCC manual as a whole.
   - [GNU C: 1.2.3] refers to Section 1.2.3 of GNU C.

4. Requirement levels — i.e., the terms "must", "should", "may", etc. — are used according to RFC 2119 [13] within this document.

5. Finally, substantial parts of Chapter 3 (excluding Section 3.5) and Chapter 6 are covered by the following publication: A. Bernauer and K. Römer, A Comprehensive Compiler-Assisted Thread Abstraction for Resource-Constrained Systems, In Proceedings of the Conference on Information Processing in Sensor Networks (IPSN), 2013.

[1]`http://gcc.gnu.org/`

1. Introduction

Technological advances in the last few decades have continuously enabled new applications of information technology due to always smaller, cheaper, more robust, and more capable devices. One central vision that emerged from this development is Smart Dust, where cubic-millimeter devices constitute “autonomous sensing, computing, and communication systems” that form “massively distributed sensor networks” [143]. Smart Dust was found unfeasible with the technology at the time. The general idea of developing and deploying *wireless sensor networks* (WSN) to autonomously monitor spatially spread physical or environmental conditions has, however, been pursued ever since. Today’s WSN applications range from wildlife monitoring [34, 83] and environmental monitoring [137, 52, 145], over control systems [16, 17], structural monitoring [67, 149], and military applications [3, 91, 124], to logistics [11] and many more [96, 118]. This thesis uses the WSN domain as an example to engage in abstractions to simplify the development of applications for resource-constrained systems.

In this introductory chapter the domain of wireless sensor networks will be introduced in Section 1.1. Section 1.2 then gives a somewhat unusual interpretation of Moore’s Law in the WSN domain, which implies a resource scarcity with various implications such as turning trivial and standard solutions to known problems infeasible. Section 1.3 first gives a brief overview of these implications in general and then focuses on one particular case, the event-based programming paradigm. Thread-based programming as an alternative to the event-based model is presented in Section 1.4, including a discussion on why it is still perceived as impossible to realize in its entirety on WSN devices. Section 1.5 then summarizes the contributions of this thesis towards overcoming the perceived obstacles and providing a comprehensive thread abstractions on resource-constrained systems nevertheless.

1.1. Wireless Sensor Networks

Although there is no strict definition of what exactly constitutes a wireless sensor network, most WSN applications share some common characteristics. First, as the term already suggests, the employed devices, so-called *motes*, are basically embedded computers equipped with sensors that allow monitoring phenomena of interest. The motes operate without wires in order to avoid the need of expensive pre-installed infrastructure and to enable mobility. This implies that they communicate via radio and have a self-sustaining energy supply via batteries or energy harvesting [106, 120].

A second common aspect of WSN applications is the inaccessibility of the motes. This is the case when, for example, human presence would disturb the monitored animals and falsify the experimental results [83], or motes are attached to wild-living badgers that stay below ground most of the time [34], or they are mounted on secluded rocks and escarpments in the Swiss Alps [52], or they are shipped inside of containers along with fruits or pharmaceuticals whose micro-climatic conditions have to be monitored [11].

Mode	Current	Mode	Current
CPU		**Radio**	
Active	8.0 mA	Rx	7.0 mA
Idle	3.2 mA	Tx (-20 dBm)	3.7 mA
ADC Noise Reduce	1.0 mA	Tx (-19 dBm)	5.2 mA
Power-down	103 μA	Tx (-15 dBm)	5.4 mA
Power-save	110 μA	Tx (-8 dBm)	6.5 mA
Standby	216 μA	Tx (-5 dBm)	7.1 mA
Extended Standby	233 μA	Tx (0 dBm)	8.5 mA
Internal Oscillator	0.93 mA	Tx (+4 dBm)	11.6 mA
LEDs	2.2 mA	Tx (+6 dBm)	13.8 mA
Sensor Board	0.7 mA	Tx (+8 dBm)	17.4 mA
EEPROM access		Tx (+10 dBm)	21.5 mA
Read	6.2 mA		
Read Time	565 μs		
Read	18.4 mA		
Read Time	12.9 ms		

Table 1.1.: *Power model for the Mica2:* The mote was measured with the micasb sensor board and a 3 V power supply [123].

In addition, the number of employed motes tends to be high, for example because a large spatial region has to be covered [137, 3, 67], or because the number of individually monitored objects is high [11], or because of required redundancy [91]. In such scenarios, networks of 64 [67], 90 [3] or more motes are not uncommon. The Argo project [137] deploys as much as 3000 individual sensors to cover the world's oceans.

Furthermore, the total system lifetime of wireless sensor networks can easily exceed several years (e.g., [137, 16, 17, 52, 67, 149]). Also, the mote usually has to be small and lightweight to meet law-regulated weights limits on animals [34], to reduce costs [11], to be unobtrusive [149], or to be hidden [3].

1.2. Moore's Law

The inaccessibility of motes makes maintenance very expensive, sometimes even impossible. This implies that the complete system lifetime has to be covered by batteries, energy harvesting, or a combination of both. Table 1.1 lists the energy demands for various operations of the Mica2 mote, a typical WSN device. The power density of today's energy harvesting technologies is in the realm of several μW to a few mW per cm^3 [106]. In consequence, even with a reasonably sized harvesting unit, only a few data packets can be received and sent per day, and batteries are required when higher mote activity is needed.

However, while memory space and system clock speed have complied with Moore's Law and exhibited an exponential growth over time, the energy density of batteries has not followed such a trend. At the same time, growing computational resources increase the overall energy consumption of a mote [69]. This keeps the physical resources of motes rather limited and makes energy saving a major design goal in the WSN domain. In addition, owing to the need of a large number of motes, a reasonable price per unit is crucial in order to keep the overall system costs within bounds. This also implies that

Mote	Mica [23, 54]	TelosB/TMote Sky [105]	TinyNode [26]
CPU	ATmega128, 7.3 MHz	TI MSP430, 8 MHz	TI MSP430, 16 MHz
RAM	128 kB	10 kB	8 kB
ROM	512 kB	48 kB	604 kB
Applications	[3, 34, 67, 83, 124]	[17, 67, 145]	[16, 52]

Table 1.2.: *Constrained resources:* Typical WSN devices do not provide much computational resources.

each device has limited physical resources, as more capable devices are more expensive. Overall, "Moore's law has an unorthodox interpretation here: it is applied toward reduced size and cost, rather than increase in capability, therefore, the amount of available physical resources is not expected to change as the technology advances" [119].

Table 1.2 shows the available resources of typical motes. It can be seen that conventional WSN hardware has a single *central processing unit* (CPU) with a clock speed between 8 MHz and 16 MHz, 8 kB to 128 kB of *random access memory* (RAM), and 48 kB to 604 kB of *non-volatile programmable memory* (ROM). As a comparison, Apple's latest iPhone 4s runs a dual core processor with a 1 GHz clock each, has 1 GB of RAM and up to 64 GB of non-volatile memory. Consequently, this smartphone costs between $649 and $849 [`store.apple.com`, 05/01/2012] and has to be recharged every other day. In contrast, motes cost between $20 and $150 [`moteware.com`, `redwirellc.com`, 05/01/2012] and can operate for months and even years without recharging.

1.3. Implications of Resource Scarcity

The scarce resources of motes have many implications on design and implementation of WSN *operating systems* (OS), protocols and applications. For instance, some devices harvest energy from the physical environment [106, 120] to account for the limited capacity of current batteries. Furthermore, radio use is a major energy drain, which resulted in the development of numerous different medium access protocols with various trade-offs and properties [97]. Other examples of energy saving include low-power listening techniques [104], in-network aggregation [81, 148], duty cycling via precise time synchronization [117], and efficient data collection [44] and dissemination [77] protocols. The limited resources also impact the operation of wireless sensor networks, as detecting faults in the presence of very limited communication is hard. Consequently, several approaches exist to engage this fundamental problem (e.g., [19, 107, 110, 111, 113, 116]).

Concerning how WSN applications are written, the constrained resources have led to the event-based programming paradigm being the predominant programming approach, mainly because of its efficient implementation. Major WSN operating systems such as TinyOS [55] and Contiki [30] have adopted this paradigm. TinyOS even employs a special programming language called nesC [42], which compiles to plain C and enforces component-based software architectures with exclusively asynchronous interfaces. For a rather long time, this has generally been perceived as a good design choice, also because the reactive nature of event-based programming is well suited for time-critical sensing applications.

In recent years, however, WSN applications have advanced from initially very simple sleep-sample-send data collection trees to more complex distributed systems with peers running IP stacks [61], HTTP and CoAP services [70], middleware [22], and business logic [95]. As a result, today's WSN applications include complex control and data flows that do not fit well with event-based programming. In fact, long causal chains of events, handlers, and subsequent events are needed to model the complex control flow with asynchronous functions. Also, data flowing across different event handlers has to be managed manually. It has been observed that this often leads to confusing, hard to manage, and error-prone code (e.g., [2, 31, 68, 88, 99, 119, 144]). We agree with this observation.

Figure 1.1 shows three basic examples of the previously discussed consequences of event-based programming. On the left appears the code written when threads are available. The `operation` is a synchronous function, and execution continues with the next statement as soon as this function returns. The equivalent event-based code is shown on the right, where `operation` is an asynchronous function which is only triggered, while the continuation is registered and called back eventually. In real applications, virtually any combination of the shown control flow examples can be found. The number of necessary continuation functions and manually managed global state increases significantly with the complexity of an application. This leads to error-prone code that is hard to maintain.

Experience has shown that humans are quickly overwhelmed by the complexity of reasonably sized systems. Particularly when the level of abstraction is low, software faults are the rule rather than the exception. Also, the amount of labor that is required to create a prototype, and more so a production-ready application, is high. It also means that development and testing is time-consuming and hinders the quick adoption of new ideas, insights, or requirements. During deployment, detecting software faults and system errors is cumbersome and expensive, in particular because the usually numerous motes are often spread out and inaccessible. And during operations, anybody within the physical vicinity of the motes can communicate with them and might be able to take over control by exploiting a software fault. Security failures can become fatal, especially when motes cannot only sense the physical environment but also affect it by means of actuators. With the emergence of the *Internet of Things* (IoT) [86], where mote-like devices are connected to the Internet and interact with the real world, security and safety of resource-constrained systems even affects the general public.

It seems therefore to be advisable to use programming models with a high level of abstraction to prevent software faults in the first place. Hence, various programming abstractions for wireless sensor networks have been proposed over the last few years [96]. They range from extending the present programming language with features to iterate over neighboring nodes and to access their remote state [48], to introducing a new domain-specific, declarative and functional language [82], and many points in the design space in between [96]. Although programming becomes easier with these abstractions, additional obstacles such as learning new languages, concepts and features can hamper the adoption to practice. Additionally, only a single one of these abstractions considers fault diagnostics [7], i.e., the Macrodebugger [125], a post-mortem debugger for MacroLab [126], a MATLAB[1] dialect for programming distributed WSN applications as a whole. In any other case, the missing support for fault diagnostics breaks the abstraction, as developers are faced with the event-based nature of the run-time system when executing it in the present debugger. The developer bears the burden of understanding the potentially complex implementation of the abstraction in order to perform the necessary back-mapping from low-level code observation to the initial abstract program. This may be a major reason preventing these abstractions from becoming popular, although we are not

[1]`http://www.mathworks.com/products/matlab/`

```
void f() {
    statement1;
    operation();
    statement2;
}
```

⇒

```
void f1() {
    statement1;
    operation(&f2);
}

void f2() {
    statement2;
}
```

```
void f() {
    statement1;
    if (condition) {
        statement2;
        operation1();
        statement3;
    } else {
        statement4;
        operation2();
        statement5;
    }
    statement6;
}
```

⇒

```
void f() {
    statement1;
    if (condition) {
        statement2;
        operation1(&f2);
    } else {
        statement4;
        operation2(&f3);
    }
}

void f2() {
    statement3;
    f4();
}

void f3() {
    statement4;
    f4();
}

void f4() {
    statement6;
}
```

```
void f() {
    statement1;
    for (int i=0; i<10; i++) {
        statement2;
        operation();
        statement3;
    }
    statement4;
}
```

⇒

```
int i;

void f() {
    statement1;
    i = 0;
    f2();
}

void f2() {
    if (!(i<10)) {
        f4();
    } else {
        statement2;
        operation(&f3);
    }
}

void f3() {
    statement3;
    i++;
    f2();
}

void f4() {
    statement4;
}
```

Figure 1.1.: *Consequences of event-based programming:* The control flow is spread across multiple functions and state has to be managed manually.

aware of any studies that investigate this correlation.

1.4. Thread-based Programming

In contrast to new languages and language concepts, a different research direction investigates the well-known paradigms of cooperative and preemptive threads as an alternative execution model of the C programming language and its dialects. There have been quite some efforts in the past to enable thread-based programming on motes, as threads allow for sequential computations via synchronous functions. This, compared to event-based programming, often leads to simpler and better manageable code. Traditionally, this has been achieved by preallocating one stack for each thread and by introducing a software *scheduler* that switches between different threads at run-time. Some WSN operating systems like Mantis OS [10] and LiteOS [15] have such a scheduler included in the kernel, while for others this functionality is implemented as a dedicated library that is executed by the event-driven OS core [68, 88, 99, 144].

In either case, this approach is not resource efficient for two reasons. First, each stack has to provide enough cut-off to support the worst case memory demand of occurring interrupt handlers operating on the respective current stack. Second, the aggregated size of all preallocated stacks is usually higher than the actual maximum stack consumption of an application, because usually not all threads reach their maximum stack usage at the same time. As a consequence, multiple, preallocated stacks do not scale well with the number of threads. Given limited RAM, this severely limits the maximum number of threads per WSN application.

Therefore, compiler-assisted thread abstractions are gaining momentum [31, 89, 119]. In this approach, a thread-based program is translated into an event-based one that can be executed by an efficient event-based kernel. Conceptually, there are three major reasons why compiler-assisted thread abstractions can be more efficient than run-time-based approaches. First, compiler-assisted thread abstractions generate code that uses the existing multi-tasking capabilities of the operating system. Run-time-based solutions, in contrast, employ a *scheduler*, which is an additional software component that performs *context switching*, i.e., suspending the execution of one task and resuming the execution of a different one. Second, compiler-assisted thread abstractions avoid multiple preallocated stacks, as the event-based run-time system operates on the single hardware stack. This results, as previously mentioned, in more efficient RAM usage. And third, compilers can perform static analysis and generate specialized code that exploits application-specific properties. Run-time-based solutions, on the other hand, always have to support the most generic case.

The first and most prominent example of a compiler-assisted thread abstraction is Contiki's protothreads [31]. This system uses the C preprocessor in a creative way to provide the syntactic illusion of threads. Although protothreads are widely used in Contiki, as they have been shown to be almost as efficient as native event-based programs, the provided thread abstraction is very limited. Among other issues, there is no automatic memory management and yield points may only exist at certain places. Worse, invalid code is sometimes not detected by the compiler, leading to run-time errors. At the same time, when investigating application faults, software developers are faced with the event-based nature of the run-time system. This breaks the abstraction and forces developers to understand the details of protothreads' implementation.

Other approaches to compiler-assisted threads employ dedicated compilers and have improved the provided thread semantics. But none of them have fully exploited the capabilities of compiler-assisted thread abstractions and they do no support fault diagnostics. With our work, we are aiming to achieve both.

1.5. Contributions

This thesis proposes the first comprehensive compiler-assisted thread abstraction for resource-constrained systems. We offer a full-fledged cooperative threading model. Also, *Ocram*, our compiler implementation, generates event-based code that can run atop any event-based kernel such as Contiki or TinyOS, only requiring a thin platform abstraction layer. Ocram detects violations of the remaining constraints of the threading model reliably, and the resulting code is almost as efficient as hand-written event-based code. Furthermore, generated and hand-written event-based code can seamlessly be integrated into each other.

Additionally, *Ruab*, our debugger implementation, completes the abstraction by enabling fault diagnostics on the abstraction level of threads, thus hiding the event-based nature of the run-time system entirely. Our approach is mostly intended for software in the service and application layers atop the operating system and its drivers, where the highest increase in complexity has been observed in the past and is expected in the future. On these layers, timing issues are of lesser concern, which is why the inability of cooperative threads to guarantee timings [68] does not impede the goal of this work, which is simplifying the creation of WSN applications.

In summary, our contributions are:

1. A platform-agnostic source-to-source transformation scheme that translates C programs using cooperative threads and synchronous functions into equivalent C programs using events and asynchronous functions,
2. Ocram, a compiler prototype that implements the transformation,
3. Ruab, a debugger prototype for Ocram that enables fault diagnostics on the thread-level of abstraction,
4. platform abstraction layers to bind Ocram to Contiki and TinyOS,
5. an extensive evaluation, which
    a) shows the feasibility of compiler-assisted threads for three different WSN application archetypes,
    b) verifies the correctness of the transformation, and
    c) measures the resource costs of this abstraction compared to both native event-based implementations and run-time-based solutions.

The complete source code of both Ocram and Ruab is published [6] under GPL 2.0 [41]. Also, the complete evaluation including all case study applications and any additional software is published the same way. This additional software includes a thread library for Contiki, a COOJA [38] plugin to profile MSP430-based applications, a Haskell [4] implementation for the GDB [129] Machine Interface, and other components. We have chosen to publish the source code of this thesis so that our claims can be verified independently, details can be looked up, and future research can be built upon it.

1.6. Structure

This thesis is organized as follows: Chapter 2 establishes the context. In Section 2.1 the terminology used throughout the thesis is clarified. WSN operating systems, both run-time-based and compiler-assisted approaches to thread abstractions, as well as various approaches to investigate faults in WSN applications are presented in 2.2. Finally, Section 2.3 provides a motivation for as well as an introduction to Haskell, the programming language we have chosen to implement the prototypes with.

Chapter 3 presents the first of two conceptual cores of this thesis: translating threads (*T-code*) to events (*E-code*). Section 3.1 gives an overview of the compiler and its principles. In 3.2, the scheme of translating thread-based code into equivalent event-based code is explained. A definition of the term "equivalent" and an informal reasoning why the translation scheme indeed generates equivalent code can be found in 3.3. Section 3.4 then explains how generated code interoperates with the operating system and existing event-based code. A description of the various stages of the compiler pipeline concludes this chapter (Section 3.5).

Chapter 4 presents the second conceptual core of this thesis: mapping the execution of the event-based system back to the abstraction of the thread-based code to facilitate fault diagnostics. In Section 4.1, the user-facing interface and concepts of the debugger are discussed. Section 4.2 explains what information the T-code debugger needs and how the T-code compiler provides this information. 4.3 elaborates on the interface between our T-code debugger and the native E-code debugger. Finally, Section 4.4 shows how the three interfaces discussed in the previous sections are integrated in order to establish the T-code debugger.

Chapter 5 provides a documentation of Ocram, a T-code compiler prototype (5.1), and Ruab, a prototypical T-code debugger (5.2). Their respective software architecture is explained, major interfaces are specified and a few selected challenges of the implementation are highlighted in each of these sections.

Chapter 6 covers the evaluation of our approach. In Section 6.1 the setup of the performed experiments is specified. Several verification mechanisms to ensure the soundness of the experimental results are discussed in 6.2. Section 6.3 explains the measured parameters. The results are presented in 6.4, followed by an interpretation in Section 6.5. This final section ends with a discussion on the limitations of the translation scheme and the prototypes, possible improvements and alternatives, and a general outlook on future research directions.

Chapter 7 concludes this thesis.

2. Background

In this chapter the context of this thesis will be established. Section 2.1 explains the intuition of operational semantics of event-based and thread-based programming, and clarifies corresponding terms used throughout this thesis. Section 2.2 presents the state-of-the-art in providing thread abstractions for, and investigating faults of, WSN applications. Finally, Section 2.3 motivates our choice to use Haskell to implement our prototypes and introduces three major language features.

2.1. Terminology

There are many slightly different definitions of the terms "thread," "event," "task," etc. in relevant literature. To avoid confusion, our definitions of the terms used throughout this thesis are clarified. Additionally, we will describe the intuition of operational semantics of thread-based and event-based applications as we use them here. We will also explain the various terms involved in fault management of applications.

2.1.1. Operating Systems and Applications

An operating systems is a set of basic software components that manages device resources and provides common services by means of an *application programming interface* (API). An API is a set of functions. Calling such a function triggers an *operation* of the OS, which, for instance, can involve reading a sensor value, sending a network packet, or writing to non-volatile memory. In case of a *synchronous function*, the associated operation is guaranteed to be completed when the function returns. In contrast, calling an *asynchronous function* triggers an operation that will only complete eventually. Signaling completion of such an operation can be done in various ways. The next section will address one of them, which is sending an event to the application.

An application usually consists of multiple *tasks*, which are logical groups of computations that pursue a common goal. Examples include tasks for continuously reading sensor values, participating in the routing or data dissemination protocols, keeping track of network neighborhoods, etc. Operating systems need to schedule multiple tasks to a single CPU while maximizing its utilization. This can be reportedly achieved most efficiently by having asynchronous OS APIs, and following the event-based paradigm.

2.1.2. Event-based Programming

An event-based system consists of a set of possible events and associated event handler functions. Conceptually, an *event* represents the occurrence of something that should provoke a reaction of the system. Examples include the reception of a radio message, a timeout, or the completion of an operation. The OS manages all current events in a possibly prioritized *event queue* and runs a *dispatcher* that repeatedly takes an event from the queue and passes it to the associated *event handler function*. In response to an event, a handler function can perform computations, trigger operations, and create new events.

Within this paradigm, a task is formed by a causal chain of events and event handler functions, frequently initiated by recurring events like timeouts or incoming radio messages. Such tasks can naturally be executed in an interleaved fashion on a system that is neither parallel nor preemptive, as the single dispatcher waits for a handler function to return before processing the next event in the queue. Thus, a single stack is enough to execute all tasks virtually in parallel. Also, task scheduling is as easy as managing the event queue and there is virtually no context switching overhead.

Although efficient, the event-based paradigm only provides little abstraction and entails high complexity. A main source of complexity is the difficulty of managing the control flow of a task, as the code is spread out amongst multiple event handlers (cf. Figure 1.1). A second major source of complexity is that execution contexts of tasks have to be manually managed and preserved between subsequent event handler invocations. The only reasonable way to do this is by using *global variables*, i.e., objects with *static storage duration* [C99: 6.2.4-3], and identifiers that have *file scope* [C99: 6.2.1-4] and *external* or *internal linkage* [C99: 6.2.2].

Additionally, every function that calls an asynchronous function becomes asynchronous itself, which recursively applies to the whole call stack and forces all functions' implementations to be split up (cf. "stack ripping" [2]).

Overall, event-based programming requires a lot of cumbersome and error-prone manual work by the programmer. This, as already discussed in Section 1.3, can lead to software faults, which make WSN applications expensive to develop, deploy and maintain, while representing a high risk for security issues.

2.1.3. Thread-based Programming

Thread-based programming overcomes the above-mentioned problems via synchronous OS APIs. With synchronous functions, the control flow is sequential, and a task's context can be stored in so-called *local variables*, i.e., objects with *automatic storage duration* [C99: 6.2.4-4–5], and with identifiers that have *block scope* [C99: 6.2.1-4] and *no linkage* [C99: 6.2.3-6]. These variables have a scope-based *lifetime* that is managed automatically.

Calling a synchronous function implies waiting for the associated operation to complete. Other tasks should therefore progress in the meanwhile to utilize the CPU. This can be realized by having one thread per task.

A *thread* is one flow of control starting with the invocation of a *thread start function* and sequentially executing the statements of that function. This includes calls to other functions and access to both local and global variables, as well as to other objects. While a thread ends as soon as its start function returns, such functions usually have an infinite loop to keep the corresponding task running.

Threads appear less expressive than events, as they can only wait for the completion of exactly one operation at any point in time. With help of the underlying platform API, however, threads can achieve equal expressiveness as events. For instance, if there is a synchronous API function `receive_with_timeout`, a thread can simultaneously wait for a timeout and an incoming network packet. This introduces complexity to the OS and could quickly increase the number of required API functions. Our evaluation in Chapter 6 shows that fortunately only a few of such combining functions are needed in practice.

Threads are considered inefficient because the context of a task is the complete stack of its thread. Determining the maximum stack size of an application is in general an undecidable problem, particularly because of recursive functions. Even when recursion and other disruptive features are removed the problem is still considerable:

The predominant execution environment on CPUs comprises a *hardware stack*, i.e., a single stack data structure for which the hardware offers special instructions. In such an environment, multi-stacked execution can be achieved by altering the hardware register that points to the beginning of the stack in memory (i.e., the *stack pointer register*). However, interrupt handler functions are not aware of this and execute on the respective current stack, assuming it to be the hardware stack. Additionally, aside from temporary interrupt masking, interrupts can both occur at any point in time and interleave with each other.

Best practice is to add enough cutoff to each stack in order to void out-of-stack situations for any possible combination of invocations of interrupt handlers [88]. This approach does not scale well with the number of threads. Also, cutoffs of a reasonable amount of threads quickly exceed the total amount of available RAM on sensor nodes.

In addition, all stacks accumulated are usually larger than the maximum of the thread's individual stack usage over time. This happens because usually not all threads reach their individual maximum at the same time. Thus, multiple preallocated stacks are not efficient with respect to RAM utilization.

Overall, these are the main reasons why providing a comprehensive thread abstraction for resource-constrained WSN devices is not trivial.

2.1.4. Preemption vs. Cooperation

There are two main ways to implement thread-based multitasking. The more common *preemptive threads* work with a scheduler that can preempt any thread at any point in time and perform a context switch. This approach guarantees timings, priorities, and fairness irrespective of the concrete implementation of each thread.

The major downside is that "[preemptive] threads [...] are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism" [74] by defining *critical sections* with mutexes and other synchronization tools. Furthermore, practice has shown that "humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code" [130].

In scenarios that we target, i.e., application and service layers of WSN applications, we encounter requirements that do not mandate preemptive threading. First, real-time is less of an issue. Second, untrusted and therefore uncooperative code is usually not included. This is why we argue for *cooperative threads*, the simpler thread programming model.

In cooperative threads, the set of functions can be partitioned into three categories. First, a *blocking function* is a synchronous function of the OS API that involves waiting for an operation to complete and blocks the calling thread. Second, a *critical function* is a function of the application that contains at least one code path that leads to the invocation of a blocking function. And third, an *auxiliary function* is a function that is neither a blocking nor a critical function. These functions may be part of either the application or the OS API.

With cooperative threads, all but one are always blocked and context switching between them can only occur when the running thread invokes a blocking function. These *yield points* form the boundaries of implicit critical sections, meaning that the use of a shared state is generally safe and has to be checked only after a *critical call*, i.e., a call of a critical or blocking function.

The execution of preemptive threads is inherently nondeterministic, as the scheduler can interrupt a thread at any point in time. Although cooperative threads clearly mitigate this, their execution is nondeterministic nevertheless, because each one of all waiting threads could continue first. If the exact order of threads is important, thread synchronization primitives are required and the OS API has to provide them.

2.1.5. Expectations on Thread Abstractions

Programmers usually have a set of expectations in relation to threads. In this section they will be stated to prepare the argument that many existing WSN thread abstractions do not meet expected standards.

A program with cooperative threads includes a set of application functions and a set of OS API functions. Some of the application functions are thread start functions, which execute threads either statically or dynamically. With *static thread creation*, threads are statically assigned to thread start functions and are started as soon as the program starts. *Dynamic thread creation*, in contrast, is able to start new threads at run-time by calling special OS API functions. The maximum number of threads is conceptually unbound and only limited by the amount of available computational resources.

Implementing critical functions should not differ from implementing auxiliary functions. They should not vary in their call semantics either, with the exception of critical calls possibly leading to a context switch. Some implementations make a syntactic difference between critical and non-critical calls to keep the developer aware of possibly changed shared state. Unfortunately, the only reasonable and C99-compliant way to do so is enforcing a naming scheme such as a fixed prefix for critical and blocking functions.

Staying compliant with C99 (or other C standards) is important because it allows using existing tools such as *integrated development environments* (IDE), static analysis tools to verify coding conventions and detect faults, etc. Thus, thread abstractions should avoid syntactic extensions to the programming language.

Threads should be able to communicate freely with each other via shared storage such as objects with static or *allocated storage duration* [C99: 7.20.3] and identifiers with file scope. All application functions should also be able to freely call each other, which implies two things in particular: First, critical calls may occur anywhere in the code. Second, any function can be *re-entrant*, i.e., multiple threads can call it at the same time, while each thread has its own set of local variables.

The expectations on preemptive threads are very similar to those on cooperative threads, except for the timing of context switches. In any case, there should not be major deviations from well-known execution semantics of preemptive or cooperative threads. This would demand for an understanding of all its implications or could lead to subtle software faults otherwise.

2.1.6. Debugging

The term *debugging* has a colloquial origin but is well established in today's software engineering domain. It refers to the process of finding a *fault*, a defect in the system under observation. This process is also referred to as *fault diagnostics*. A fault can lead to an *error*, which refers to "that part of the system state that is liable to lead to subsequent failure” [50]. In contrast, a *failure* “occurs when the delivered service no longer complies with the specification” [50].

A *source-level debugger* is a controlled execution environment for the system under observation that offers debugging tools to users. First, it makes the system state observable and sometimes even modifiable. Second, it offers *breakpoints*, i.e., stopping system execution when user-defined conditions are met. The important aspect hereby is that the user’s view on the system is its source code, i.e., variables and language statements. Details of how the source is compiled and how the binary is executed are hidden, enabling users to focus on system faults.

During fault diagnostics, the user tries to reproduce an observed failure by bringing the system back to the point where the failure occurs. By inspecting the system state it is then possible to find the error and deduce the fault from this information.

2.2. State-of-the-Art

Wireless sensor networks enable important applications, but they are extreme with respect to scarcity of resources. This domain serves as a good environment to develop and evaluate our approach. Nevertheless, Section 2.2.1 will first address a general discussion on multi-tasking to establish the subsequent discussion that is focused on wireless sensor networks.

WSN research has been conducted for over a decade now with many results produced. We have chosen to discuss the following subset that we consider particularly relevant for our work. Our approach is based on event-based run-time systems, which is why WSN operating systems are presented in Section 2.2.2. Thread-based WSN operating systems are also discussed briefly, as they target the same problem as our approach does. Section 2.2.3 provides a brief survey of WSN thread libraries and their various techniques, trade-offs, and compromises with respect to supported thread semantics.

In Section 2.2.4 existing compiler-assisted thread abstractions are examined by following their evolution and explaining why we still consider them to be incomplete. Nevertheless, our work builds upon these ideas and adds the necessary steps to obtain a comprehensive compiler-assisted thread abstraction for resource-constrained systems. Finally, some suggested approaches for debugging in wireless sensor networks are outlined in 2.2.5.

2.2.1. Multi-tasking

The core of the problem our work targets steams from the need to support multiple tasks within a single program. This section discusses various known solutions to this problem and puts our approach in perspective.

Scheduling a number of tasks on a lower number of CPUs requires some means to suspend and resume the execution of individual tasks. Section 2.2.1.1 is concerned with the state of a task which has to be stored and reloaded to this end. Subsequently, Section 2.2.1.2 provides a short introduction to coroutines, a very expressive concept which enables various control structures. Amongst them, thread-based multi-tasking can be regarded as a special case of coroutines. Finally, Section 2.2.1.3 provides a tight survey on the long-standing argument on threads vs. events.

2.2.1.1. Continuations

Suspending a task and resuming it later requires the manifestation of the task's state as an ordinary data structure that can be managed in a wait queue for instance. Such a manifestation is generally referred to as the *continuation* of that task. Different run-time system have varying definitions and implementations of continuations. We will cover the most important ones in this section.

On the level of assembler languages, a continuation consist of an instruction and a stack pointer along with the assumption that the referenced code and data is preserved. Software engineers in this domain are used to deal with continuations explicitly. In fact, every function call and return entails copying stack and instruction pointers between registers and the stack, so that the call stack of a task contains many continuations, one for each pending function call. Such a continuation does not only determine where to continue the execution of a task, but it also references the state of all local variables as stored on the corresponding stack. In contrast, other aspects such as allocated memory and file resources are not covered and have to be managed manually as needed.

The concept of continuations is generic enough to be viable not only for function calls and returns. For instance, performing a *non-local jump* that crosses multiple stack frames is trivially possible and can be used to implement exceptional control flows. The use case most important in the context of this thesis is the implementation of thread-based multi-tasking by managing continuations referring to different stacks [88].

C99 provides an abstraction for a restricted type of continuations. The function (or macro) `setjmp` "saves its calling environment [i.e., its continuation] in its [...] argument for later use by the `longjmp` function" [C99: 7.13.1.1]. Invoking `longjmp` performs a non-local jump to the point after the corresponding call to `setjmp`. The continuation is stored in an array which can be arbitrarily managed, thus enabling a variety of different control structures.

C99 continuations are restricted because `longjmp` is not required to preserve the current execution environment. Jumping into a function that has previously been left via `longjmp` therefore results in undefined behavior. This situation, however, occurs every time a previously suspended task is resumed. Implementing thread-based multi-tasking in C99 is therefore not portable, but nevertheless possible with many implementations.

A C99 continuation in event-based programming is typically limited to only carry an instruction pointer while the stack pointer is assumed to be the same for all continuations. While this enables a very efficient implementation of multi-tasking it excludes local variables. When resuming a continuation pointing to the middle of a function, the state of local variables is usually lost and reset to some undefined value, as the execution of other tasks has invalidated the corresponding stack frame in the meanwhile. Protothreads follow this approach, thus inducing treacherous run-time behavior of the system (cf. Section 2.2.4). To avoid confusion, most event-based systems therefore restrict a continuation to refer to the start of a function where all local variables have not been initialized yet anyway.

In other words, the instruction pointer of a continuation is obtained by retrieving the address of a *call-back function*. This address is either registered in advance or passed along while yielding, constituting an imperative continuation-passing style (Section 3.4 of [33]). The downside is that there must be a distinct function for each continuation of a task, resulting in the control flow being split and spread over a possibly large number of functions. Also, local variables still are not part of a continuation, which disables the usage of automatic storage and forces manual memory management instead.

In order to translate a thread-based program into an event-based one we basically have to transform thread-style continuations with arbitrary instruction and different stack pointers into event-style continuations with addresses of call-back functions and a single implicit stack pointer. Chapter 3 will explain how computed gotos and statically allocated data structures resembling run-time stacks compensate for the limited event-style continuations and enable such a translation.

Languages like Scheme [33] address continuations by elevating them to first-order objects [142]. The function `call/cc`[1] takes a single function `fun` which is called with the current continuation. Such a continuation is itself a function taking a single parameter `res`. As functions are first-order objects in Scheme, a continuation can be stored and arbitrarily managed by a scheduler for instance. If `fun` returns, `call/cc` returns with the result of `fun`. Whenever the continuation is called, the current control flow is aborted and execution resumes as if the corresponding call to `call/cc` had just returned `res`.

A Scheme continuation can be resumed arbitrarily often. Associated closures additionally manage all referenced data automatically, turning this facility into a very powerful and generic tool for various control structures. Non-local jumps (a.k.a. non-local exits) are trivial and provide a comfortable way to handle exceptional control flows for functional programs. Continuations can also be used to implement threads [122], where suspending a task is calling the scheduler function via `call/cc` and resuming a task simply involves calling the received continuation. Because continuations are so flexible, they even enable the implementation of coroutines [53], which we will discuss in the next section.

This flexibility comes at a cost, though, as it requires a solution to the *funarg problem* [94]. This problem emerges whenever a function references variables from its outer environment, for instance a local variable of one of its callers. When that function is invoked later as part of a continuation, the surrounding environment might have changed, e.g., the original caller might have returned and the associated stack frame has therefore been invalidated. Possible solutions vary from switching to allocated storage and utilize either garbage collection or reference counting [4], to limiting support to read-only variables and to copy them [45]. In any case, up to our knowledge, no solution is efficient enough to enable Scheme-style continuations on resource-constrained system like the ones we target.

2.2.1.2. Threads and Coroutines

Coroutines [84] are a generalization of subroutines (a.k.a. functions). While functions can only be exited once, coroutines can exit (i.e., yield) and resume arbitrarily often. Whenever a coroutine resumes it behaves as if it had never exited, only that shared state might have been modified in the meanwhile.

When a coroutine yields, it names the coroutine which should resume next. Although this looks like a normal function call at first, it is important to note that the invoked coroutine can yield back to the original coroutine. In contrast to the caller-callee schema of ordinary function calls, "coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program" [21].

Coroutines represent a very interesting research topic because most if not all control flow structures can be regarded as a special case of coroutines. For instance, coroutines can express state machines within a single subroutine, actors [20], generators [133], *Communicating Sequential Processes* (CSP) [115] as well as cooperative threads. In the

[1]short for `call-with-current-continuation`, originally called `catch`.

latter case, each thread execution function is in fact a coroutine. Whenever a thread yields the coroutine yields back to the scheduler which in turn yields to the next coroutine.

Interestingly enough, coroutines can be implemented by means of cooperative threads. This simply requires one thread per coroutine and a single shared variable holding the name of a coroutine. Whenever a coroutine yields, its thread updates this variable with the name of the coroutine which should resume next. The scheduler respects this choice and invokes the thread corresponding to the requested coroutine next.

While this suggests that implementing coroutines is as expensive as implementing threads, the former is a rather uncommon multi-tasking model in C. The reason for this might be historic, as originally the C language family did not even cover threads. Instead, libraries like pthreads (IEEE Std 1003.1c-1995) and GNU Portable Threads [36] introduced preemptive and cooperative threads after the fact, while a comparable effort is unknown to have been attempted for coroutines. In order to provide a familiar programming abstraction as an alternative to event-based multi-threading, our work introduces an implementation of threads as opposed to coroutines.

2.2.1.3. Threads vs. Events

There has been an ongoing debate in the research community about whether event-based or thread-based programming is superior. For instance, Ousterhout argued that "threads are a bad idea" [102] and von Behren et al. suggest that "events are a bad idea" [138]. Opinions seem to depend on requirements and users preferences. In general, both concepts are known to be dual to each other [73]. Adya et al. even managed to "exhibit adaptors that enable automatic stack management [thread-based] code and manual stack management [event-based] code to interoperate in the same code base" [2].

Opposing parties have been working on their favorite paradigm to mitigate respective disadvantages. For example, von Behren et al. argue that "a modern thread package will be able to provide the same benefits as an event system while also offering a better programming model for Internet services" [139]. To support this claim, they introduced Capriccio, a scalable thread package for cooperative threads on Linux that comes with resource-aware scheduling. To solve "the problem of stack allocation for large number of threads", the authors employ a compiler that augments the application with code for dynamic stack management.

Overall, however, "[the authors] believe there is no advantage to a static transformation from threaded code to event-driven code, because a well-tuned thread run-time can perform just as well as an event-based one" [139]. From today's perspective, either past WSN research has failed in finding a "well-tuned thread run-time" or the previous statement does not apply to wireless sensor networks. In fact, run-time-based WSN thread abstractions either provide incomplete thread semantics or are clearly less efficient than existing event run-time solutions (Section 2.2.3). We believe that resource scarcity makes a significant difference to other domains and static transformation therefore is the key to build a comprehensive thread abstraction.

For event-based programming, Cunningham and Kohler introduced the Explicit Event Library (libeel) to make "event-driven code easier to read, write, debug, and maintain“ [24]. Overall, libeel "sustains the advantages of event-driven programming while adding the important advantage of programmability", which is achieved by designing libeel to be "amenable to programming analysis", and by creating "tools to graphically expose control flow, verify resource safety properties, and simplify debugging". Although libeel improves event-based programming, control flow is still split among multiple event handlers and data flow has to be managed manually. Our approach, on the other hand, includes the comfort of threads while being almost as efficient as event-based programming.

Approaching the issue from a different angle, Harris et al. introduced “AC, a set of language constructs for composable asynchronous IO” [51]. These language constructs include **`async`**, **`do`**/**`finish`**, and **`cancel`** operations that retain “a sequential style of programming without requiring code to be ‘stack-ripped’ into chains of callbacks”. Nevertheless, the run-time system is event-based and it performs the mapping between sequential code and asynchronous *input and output operations* (IO). With our work, the compiler performs the mapping at compile-time, therefore saving run-time resources.

Similar approaches have been explored by others and have even become mainstream [135]. In all these cases, new language constructs and new control flow semantics are added. Our work keeps the well-known cooperative threading paradigm, which we consider adequate for solving the addressed problems.

2.2.2. Operating Systems

For the scope of this thesis, sensor network operating systems can be divided in two categories by distinguishing operating systems that natively support threads from those that do not. The latter category will be presented first.

2.2.2.1. Event-based Operating Systems

One main WSN operating system is TinyOS [55], whose development began in the late nineties. The operating system and all applications are written in nesC [42], a specifically designed dialect of the C programming language. This language enables static analysis by disallowing pointer arithmetics and function pointers. It also enforces a component-based software architecture with exclusively asynchronous interfaces. Interface definition, component construction, and component composition are three separate aspects, which can flexibly be integrated for a given application.

There are two execution contexts in TinyOS: interrupt handlers and tasks (not to be confused with the term “task” from Section 2.1.1). Tasks are deferred procedure calls, scheduled one after the other, that always run to completion. Interrupt handlers on the contrary can be preemptively invoked at any point in time. To deal with the caused concurrency, nesC distinguishes between *synchronous* and *asynchronous context*. Tasks are synchronous context, interrupt handlers are asynchronous context, and functions can be declared to be in either. The rule is that synchronous functions cannot be called from asynchronous context. Thus, asynchronous functions are the only place where explicit atomicity has to be established as needed.

Implementing complex control flows with nesC quickly becomes difficult and error-prone, because all functions are asynchronous, causing corresponding continuations being spread all over the program. Reading code written by others and comprehending the control flow at hand is still harder. During debugging, stepping from the invocation of an operation to the point in the corresponding control flow that handles its completion is cumbersome.

A second major operating system for wireless sensor networks is Contiki [30]. The focus of Contiki was not originally on wireless sensor networks. Instead, Contiki has aimed to be small and highly portable, targeting memory-constrained networked systems in general. Today, the Contiki community calls it "The [..] Operating System for the Internet of Things"[2]. Contiki is in particular famous for its networking capabilities, including μ-IP stack [29], IPv6 stack with 802.15.4 6lowpan [93] header compression and fragmentation [32], low-power IPv6 routing capabilities [136], and CoAP implementation [70]. In fact, there is now only a thin line between Internet of Things and wireless sensor networks due to increasingly similar or identical technologies, devices, and requirements. Contiki can therefore be a good choice for sensor network applications and it is in fact used often in that domain.

The run-time system of Contiki is event-based, although some operations such as sending network packets are implemented via synchronous functions. Nevertheless, receiving network packets, waiting for a time-out, and other operations alike have to be implemented via asynchronous functions because the run-time system is single-threaded. The Contiki community recognized the problems of event-based programming and introduced protothreads, "simplifying event-driven programming of memory-constrained embedded systems" [31]. Today, protothreads are deeply integrated into the Contiki system, making its use almost mandatory. Section 2.2.4 will explain details of protothreads as they constitute the first compiler-assisted thread abstraction for resource-constrained systems. Being the first of its kind, protothreads still have some limitations. Our work aims to take necessary next steps to improve this approach.

To complete the discussion on event-based WSN operating systems, we want to mention SOS [49], which accounts for the need to reprogram remote, or inaccessible motes. "SOS consists of dynamically-loaded modules and a common kernel, which implements messaging, dynamic memory, and module loading and unloading, among other services" [49]. Although addressing a real problem and providing a good solution, SOS was not widely adopted in practice. With regard to our work, SOS could be used as an underlying operating system and have dynamically-loaded modules written with our abstraction.

2.2.2.2. Thread-based Operating Systems

To avoid event-based programming altogether, some WSN operating systems support threads natively. For example, MANTIS (multimodal system for networks of in-situ wireless sensors) "is a multi-threaded cross-platform embedded operating system for wireless sensor networks" [10]. At first, the energy efficiency of MANTIS was not competitive with event-based operating systems, but it has improved significantly and now "it is possible to make a multi-threaded sensor network operating system as power-efficient as an event-based system" [28]. Nevertheless, "the sensor network community selected TinyOS as the de facto standard" [27], which motivated porting the MANTIS technology to TinyOS [27] and discontinuing the project.

Other multi-threaded operating systems such as LiteOS [15] and RETOS [18] aim for providing UNIX-like and POSIX-compliant interfaces. Still other like Nano-RK [39] focus on reserving resources to guarantee task deadlines. A completely different approach is followed with Maté, "a tiny communication-centric virtual machine [whose] high-level interface allows complex programs to be very short [...], reducing energy costs of transmitting new programs" [78]. Like SOS, Maté addresses the need to dynamically reprogram motes and in addition its "instructions hide the asynchrony (and race conditions) of TinyOS programming" [78].

[2]`http://www.contiki-os.org`

In any case, thread-based operating systems follow a run-time based approach to threads which is why they have the same problems as thread libraries do. These will be addressed next.

2.2.3. Thread Libraries

Although event-based operating systems brought sensor network applications to smaller devices, there is still a demand for higher programming abstractions. This has led to several approaches of run-time-based thread implementations on top of event-based systems.

Welsh et al. employ two execution contexts, which they call fibers: “the default system fiber is event-driven and may not block” while “the application fiber is permitted to block” [144]. Both fibers “can share a single stack”, therefore this “thread-like concurrency model for TinyOS” is very lightweight. It is, however, also rather limited because there can only be one blocking execution context at any point in time.

Y-Threads [99] constitute a generalization of fibers. They “provide separate small stacks for blocking portions of applications, while allowing for shared stacks for non-blocking computations”. The basic observation regarding Y-Threads is that the “size of stack required to execute the control behavior is fairly small.” Y-Threads provide a dedicated API for run-to-completion functions, which can all share the same stack space because they don’t interleave with each other. Once these parts are extracted from a thread, there is only a small amount of state left. It requires its own stack and is saved there during blocking computations. The downside of Y-Threads is that programmers have to manually and explicitly extract run-to-completion parts and call them indirectly via a provided API. This again separates corresponding code fragments as it is the case with event-based programming. Additionally, evolving applications that are based on Y-Threads can cause cumbersome refactoring once a hitherto run-to-completion part now happens to involve a blocking computation. All of this reflects the non-standard thread semantics of Y-Threads.

In contrast, TinyThread [88] is “a library for TinyOS and nesC that enables true multi-threading on a mote”. In this context, the term “true” refers to what is in general meant with “multi-threading”, i.e., thread-based multi-tasking. Each thread has its own stack and the scheduler switches the machine’s stack pointer register to perform context switches between threads. The application code consists of a set of functions including some thread start functions. Besides explicitly allocating stacks and assigning them to individual threads, there is no boilerplate code necessary.

As discussed in Section 2.1.3, using multiple preallocated stacks introduces significant memory overhead. To mitigate this, TinyThread employs advanced stack size estimation algorithms that are context-sensitive and “exploit the fact that interrupts are disabled in different parts of the application” [88]. The total extra memory added to the stacks still grows with $\Theta(n \times m)$, with n being the number of threads and m being the average number of enabled interrupts. Also, the total memory used for stacks still tends to be larger than the applications’ maximum stack usage over time.

The most recent run-time-based thread abstraction is TOSThreads, “a fully preemptive threads package for TinyOS [that] resolves the tension between the ease of thread-based programming and the efficiency of event-based programming by running all event-based code inside a single high priority kernel thread and all application code inside application threads, which only execute whenever the kernel thread becomes idle” [68]. As this is a run-time-based approach, “the memory costs associated with maintaining per thread stacks can be substantial” [68]. Also, as already discussed in Section 2.1.4, we advocate cooperation instead of preemption.

Overall, existing thread abstractions for WSN applications can be divided into two categories. The first imposes limitations on thread semantics (with respect to Section 2.1.5) to account for constrained resources of WSN devices. The second supports complete thread semantics, but at the cost of inefficient RAM utilization, which requires more capable and expensive hardware. It seems that run-time based approaches cannot escape from this trade-off.

2.2.4. Compiler-Assisted Thread Abstractions

Compiler-assisted thread abstractions require a dedicated compiler that translates a thread-based program into an equivalent event-based program. The comfort of threads is combined with the efficiency of events. Conceptually, there are two reasons why this approach achieves better performance. First, the run-time system is event-based, and uses the existing event dispatcher for task scheduling and context switching. Multiple preallocated stacks are hereby avoided. Second, a compiler can exploit application-dependent properties and generate optimized code, while a run-time-based approach always has to assume the most generic case.

The first system that provided a compiler-assisted thread-abstraction for WSN applications was protothreads [31]. It has been specifically designed for Contiki and its usage is deeply embedded in Contiki’s libraries and run-time system. Technically, protothreads are a set of C preprocessor macros that enable the syntactical illusion of threads and synchronous functions. Instead of blocking until the completion of an operation that involves waiting, the run-time system memorizes the current code location and returns from the handler function. When resuming the thread, the same function is called again and a `switch` statement, expanded from the protothreads macros, brings the execution back to the location where the previous wait operation was triggered. Storing a code location implies only two bytes of overhead for each thread (a.k.a. *protothreads process*), and therefore the authors claim to provide the most efficient thread implementation.

There are two reasons for protothreads’ performance. First, the potential of compiler-assisted thread abstractions is partially exploited by utilizing an event-based run-time. Second, and more important, as the C preprocessor can only locally replace language tokens and is not a dedicated compiler, protothreads’ semantics have a number of limitations.

First, blocking functions can only be called in the top-level thread start function as opposed to nested functions. This severely restricts the software architecture of protothreads applications, as the encapsulation of repetitive code patterns that involve blocking functions is prohibited. Second, the state of local variables is not preserved across calls of blocking functions. They are reset to undefined values instead because at the thread start function actually returns at run-time and is invoked again later. This unexpected deviation from well-known execution semantics has reportedly taken many developers by surprise. Even worse, such faults have to be indirectly inferred from observing the misbehaving execution of the application, as the protothreads tool chain is not able to catch the issue. Finally, using additional `switch` statements in thread start functions can interfere with the ones expanded from the protothreads macros in unexpected ways. The protothreads tool chain again happily accepts such programs as input and generates an output program with absurd run-time behavior.

Compared to thread libraries, the thread abstraction provided by protothreads is clearly incomplete and also treacherous. Nevertheless, protothreads are widely used, which indicates that thread-based programming is needed. Our work aims to satisfy this need by providing a comprehensive and accustomed thread abstraction which is still efficient enough to be executed on resource-constrained devices such as motes.

The first system, that in contrast to protothreads, employed a dedicated compiler to provide thread abstractions for wireless sensor networks was TinyVT [119]. It was specifically designed for TinyOS, which provides a component-based software architecture with event-based interfaces. TinyVT enables software developers to implement single components sequentially, as it is possible to embed event handlers in code blocks following a special `await` statement. Also, the run-time system preserves the state of local variables across such operations.

Although TinyVT overcomes many of protothreads' drawbacks, supported thread semantics are still restricted. Embedded event handlers must not contain `await` statements, and it is not possible to split the implementation of a component into multiple functions, which implies that code cannot be shared between multiple threads and functions cannot be re-entrant.

The most recent compiler-assisted thread abstraction for TinyOS is UnStacked C, "a source-to-source transformation that can translate multi-threaded programs into stack-less continuations" [89]. This is a hybrid approach, as the compiler input is an application which uses a thread library such as TOSThreads, and it generates an improved application, which uses a modified version of the original thread library. Furthermore, the compiler replaces the preemptive computation model with *lazy preemption*, reducing overhead by softening the timing guarantees of context switching. Although supported thread semantics are rather complete, the generated run-time system still depends on a thread library and it is still unclear if lazy preemption is appropriate for WSN applications.

Overall, compiler-assisted thread abstractions have shown the capability to unify the comfort of cooperative threads with the efficiency of events. However, existing work has not fully exploited the possibilities of this approach, as they only support either limited or non-standard thread semantics (cf. Section 2.1.5) and do not exploit application-specific properties. With our work, we want to advance to the next step by introducing a dedicated compiler that:

1. offers a comprehensive cooperative thread abstraction,
2. performs a source-to-source transformation of C code,
3. rejects invalid input code,
4. exploits application-specific properties, and
5. is platform-agnostic with respect to the operating system that executes the generated, event-based application.

2.2.5. Fault Diagnostics Tools

Fault diagnostics is, like programming, a question of abstraction. For instance, reading CPU registers and single-stepping the CPU execution via JTAG[3] is possible, but may

[3]IEEE 1149.1

not be the right way to find faults in applications written in languages like MacroLab [126], for instance, as the provided abstraction is built upon distributed data vectors and computations on them. For a good reason, Macrodebugger [125], a debugger for MacroLab code, has been introduced.

The debugger should ideally operate on the same abstraction level as the compiler. For this reason YETI [14], an Eclipse [132] plugin, has been developed. It closes the gap between nesC and C by supporting debugging of nesC applications. These are, as already mentioned in Section 1.3, rare examples of debuggers that have been introduced along with a new programming abstraction.

Other WSN research efforts in this regard focus on finding faults in applications written in C or one of its dialects, whether an additional programming abstraction has been used or not. MSPsim, for example, is a "MSP430 instruction level simulator that simulates sensor boards with peripherals for the purpose of reducing development and debugging time" [37]. COOJA is a graphical network simulator that integrates MSPsim and provides a plug-in for interactive debugging of MSP430 binaries [38]. Similarly, TOSSIM provides "accurate and scalable simulation of entire TinyOS applications" [76]. Technically, TOSSIM is a Python [133] framework that enables the user to implement tests and debug TinyOS applications by writing Python scripts.

Simulating the execution of WSN applications via MSPsim or TOSSIM saves a lot of time. Debugging on real hardware is still necessary, because lab conditions can differ significantly from simulator models. Thus debugging on real hardware is necessary. Clairvoyant "is a comprehensive source-level debugger for wireless, embedded networks [where] a developer can wirelessly connect to a sensor network and execute standard debugging commands [...] as well as new commands that are specially designed for debugging WSNs" [147]. Likewise, Marionette "facilitates interactive development and debugging" by providing "the ability to call functions and to read or write variables on pre-compiled, embedded programs at run-time" [146].

Deep embedding into the real world causes lab conditions to differ greatly from deployment conditions. Therefore, debugging of deployments is also required. Passive Distributed Assertions "allow a programmer to formulate assertions over distributed node states [...] causing the sensor network to emit information that can be [...] evaluated to verify that assertions hold" [113]. Collecting this information can be done using a temporary deployment support network [9] that passively overhears network traffic. LiveNet "is based on the use of multiple passive packet sniffers co-located with the network, which collect packet traces that are merged to form a global picture of the network's operation" [19]. The SNIF framework is "a general framework for passive inspection of multi-hop sensor networks to detect problems related to individual nodes (e.g., reboot, death), wireless links, paths (e.g., routing failures, loops), or global problems (e.g., partitions)" [111]. Memento [116] and Sympathy [107] are other solutions that can detect a pre-defined set of known problems during deployment and move the scope from debugging implementations to monitoring operations.

All these technologies ignore the abstraction level used to develop the application under observation. Thus, it is a challenging task to find an actual software fault from data that has been collected by such tools. We, on the other hand, want to complete the abstraction of cooperative threads by sustaining it during fault diagnostics as well. For this reason, we introduce a source-level debugger, that

1. operates on the abstraction level of cooperative threads,

2. supports both global and thread-specific breakpoints,
3. supports the evaluation of arbitrary expressions over identifiers,
4. has a modular architecture that enables different user front-ends,
5. can be executed on any platform that has a GDB [129] target.

2.3. Haskell

Our prototypes of T-code compiler and debugger are implemented in Haskell. We want to motivate this choice by presenting previous approaches in Section 2.3.1 and by explaining some core features of Haskell in Section 2.3.2. Basic knowledge about those features is required to follow parts of Chapter 5. Covering the whole spectrum of language features goes beyond the scope of this thesis, and we refer to the literature instead (e.g., [4, 5, 80, 101]).

Haskell is a non-strict, purely functional and statically typed programming language, which was named after Haskell B. Curry. The non-strict evaluation strategy is implemented via call-by-need, a.k.a. lazy evaluation. This allows for computations on infinite data structures and, more importantly, provides a tool to cleanly separate concerns. Being pure means that the result of a function solely depends on its input arguments, i.e., it constitutes a function in the mathematical sense. This ensures referential transparency, a key property of functional programs, yielding deterministic behavior that enables reasoning about program correctness, allows optimizations such as memoization and common subexpression elimination, facilitates easy unit testing, etc. Haskell's purity particularly entails that side effects are not possible by default, but have to be made explicit via monads (Section 2.3.2.3). In fact, "one of Haskell's main impacts on mainstream programming [is believed to be] the realization that being explicit about [side] effects is extremely useful" [58]. Finally, Haskell's powerful type system provides strong static guarantees for a program while its Damas–Hindley–Milner-based type inference [25, 56, 92] reliefs software developers from specifying the involved types in most cases.

Being functional in general implies, among other things, that functions are first-class citizens, i.e., they can be arguments to or results from other functions, so-called higher-order functions. "Higher-order functions and lazy evaluation can contribute significantly to modularity. [...] Since modularity is the key to successful programming, functional programming offers important advantages for software development" [59]. Also, "compilers and other code translation are natural applications for functional languages" [58]. After all, the core of a compiler is a function from input code to output code. This function particularly has to identify sections of abstract syntax trees, either in order to collect static information or to perform transformations. Algebraic data types and pattern matching, a common feature of functional programming languages, lends itself very nicely to this purpose. It was therefore clear to us that we should use a functional programming language to implement Ocram.

2.3.1. Alternative Approaches

We first tried to connect Rats! [46], a parser generated for Java [45] that builds on recent research on parsing expression grammars [40], with Scala [100], a functional programming language for the *Java Virtual Machine* (JVM). Scala is actually a multi-paradigm programming language, as it supports Java-style object-oriented programming as well. It integrates seamlessly with Java and other languages that generate bytecode for the JVM, so using parsers generated by Rats! poses no problem. Scala is strict by default, statically typed, and supports basic type inference, algebraic data types with pattern matching via so-called case classes, and other basic functional programming features. Rats! is part of the Extensible Compiler Project (xtc)[4], which provides, among other things, a grammar for C with common GNU C extensions.

Unfortunately, the abstract syntax trees returned by Rats! parsers consist of generic tuples containing a name and a list of child nodes. To use the pattern matching capabilities of Scala, translating these tuples to instances of case classes is required. Implementing and maintaining the translation of the approximately 140 different nodes that are involved in xtc's abstract syntax tree of C seemed too cumbersome. When we tried to generate the translation code instead, we faced various time-consuming technical obstacles. Furthermore, xtc provides no pretty printer, which is something Ocram needs to generate the concrete syntax of the E-code application. So, we went on to try a different alternative.

The C Intermediate Language (CIL) [98] is "a representation that makes it easy to analyze and manipulate C programs" [98]. A front-end translates C programs to CIL, thereby simplifying many language constructs. CIL also provides "a set of tools that permit easy analysis and source-to-source transformation of C programs" [98] as well as a pretty printer. CIL's recommended way of usage is to write an OCaml module [109]. OCaml is the object-oriented variant of Caml, which is a dialect of the ML programming language family. It is strict and statically typed, supports type inference, is both functional and object-oriented, and provides a rich set of additional features. Overall, it seemed like an adequate tool for the given task and we quickly managed to write early compiler prototypes.

The problem was that the CIL framework does not maintain a log of transformations applied to the input program. This knowledge is, however, required by the T-code debugger, as it is supposed to map the E-code run-time all the way back to T-code source, as opposed to displaying the CIL representation of the code to the user. The implementation of CIL seemed to be too complex for us to be able to include this feature in reasonable time. We therefore postponed this approach to try a third alternative.

The Language.C library [57] is mostly a parser and pretty printer for K&R C [66], C99, and GNU C. To use this Haskell library, our compiler had to be written in Haskell as well. Our initial experience with Haskell in general and Language.C in particular was so positive, that we decided to continue with this approach. After finishing the implementation of Ocram, we even continued to use Haskell to implement Ruab, although a debugger with user interaction and sub-process communication is not a natural application for functional languages. Nevertheless, Haskell turned out to be a good choice for Ruab as well.

First of all, numerous classes of software faults do not exist in Haskell, simply because the abstraction prevents memory corruption and because there is no state by default. The latter reason, which originates from Haskell's purity, involves race conditions, uninitialized data, usage of destroyed objects, and other common problems of imperative programming languages. Also, the type system catches most mistakes, leading to the often stated phrase: "if it compiles, it's correct". Certainly, this cannot be always true, but experience shows that this is often the case nevertheless. And when there is a fault in the program, comparing the expected output with the real output often directly pointed

[4]`http://cs.nyu.edu/rgrimm/xtc/`

to exactly one place in the code that could possibly be the cause. This is mainly due to Haskell's expressiveness and abstraction capabilities, which allows software developers to follow the Don't Repeat Yourself (DRY) principle [62] to its extreme.

2.3.2. Haskell in a Nutshell

The history of Haskell is summarized in [58]. It goes back to 1987 when the functional programming community at the Conference on Functional Programming and Computer Architecture decided to form a committee to design a purely functional programming language with lazy evaluation. The motivation was to define a common language to provide "faster communication of new ideas, a stable foundation for real application development, and a vehicle through which others would be encouraged to use functional languages" [58]. In 1990, the first version of Haskell was defined, followed by a series of updates, which culminated in the Haskell 98 Language Report, a stable, minimal, and portable version of Haskell. Although the Haskell Committee ceased to exist in 1999, Haskell continued to be adopted by others who identified small flaws in the language design and ambiguities in the report. As a consequence, The Revised Haskell 98 Report [5] was published in 2002. In 2005, a new committee has been formed to design Haskell′ (Haskell-prime), a successor language that includes language extensions which are used in practise. The Haskell community was heavily involved in the public discussions on Haskell′, leading to the first revision in 2010 [4].

Although Haskell started as an academic playground, it is widely used in the "real world" today [101]. The *Glasgow Haskell Compiler* (GHC) [85] is an industrial-strength, cross-platform, and open source Haskell implementation, which provides many language extensions and employs advanced optimization techniques. Several companies employ Haskell as a productive tool for their business, ranging from electronic design automation to systems programming for Linux-based consumer products, artificial intelligence software for decision support, a domain-specific language for specifying cryptographic algorithms, and to a WebDAV server with audit trails and logging [58]. Haskell has also influenced other programming languages such as Python, Java, C#, Visual Basic, and Scala [58]. The Haskell community is constantly growing, creating a huge ecosystem of libraries and a plethora of learning material for beginners (e.g., [80]). Nevertheless, Haskell is still tightly coupled with recent research, continuously yielding GHC language extensions and libraries whose foundations have been published in high quality conferences and journals (e.g., [60, 71]). Our perception is that this combination of established research and pragmatic public is what drives the success of Haskell.

We will briefly describe a few key features of the Haskell programming language, i.e., polymorphic data types (Section 2.3.2.1), type classes (Section 2.3.2.2), and monads (Section 2.3.2.3). These sections provide the background for Chapter 5. For a more extensive explanation of the concepts we recommend reading [80].

2.3.2.1. Polymorphic Data Types

Haskell's type system supports polymorphic data types, i.e., data types that are parametrized with other types. For example, the type `Maybe` is defined as

```
data Maybe b = Just b | Nothing
```

and encapsulates a value of type `b` that is either present (reflected by the *data constructor* `Just`) or not (reflected by the data constructor `Nothing`). This allows to model computations that can fail, while a subsequent pattern matching on the data constructors can either obtain the returned value or handle the failure case accordingly.

A second example is the type

```
data Either b c = Left b | Right c
```

which can be used to model computations that either return a value of type `b` (`Left`) or a value of type `c` (`Right`). A common scenario is `b` being an error type and `c` being a result type. A function that for instance returns `Either String Int` can then either return an integer or signal failure by returning a string that describes what went wrong. Again, pattern matching can distinguish the two data cases to react accordingly. We will make use of `Maybe` and `Either` particularly in Section 3.5.

As a final example, the Language.C data types that are used to build up the abstract syntax tree are polymorphic with a single type parameter (cf. Appendix A). For instance, the data type

```
data CTranslationUnit a =
  CTranslUnit [CExternalDeclaration a] a
```

models a translation unit that consists of a list of external declarations and a single value of any type. This allows for annotating tree nodes with arbitrary information. We will make use of this in Section 4.2.2.1.

2.3.2.2. Type Classes

A Type class basically is a set of functions which is parameterized with a single type. A given type can implement a type class if it provides definitions for all of its functions. For example, the type class

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

states, that any type `a` can be a monoid if there is an overloaded version of both `mempty` that returns a value of that type and `mappend` that maps two values of that type to a third one. The documentation of `Monoid` clarifies, that `mappend` is a binary operation and `mempty` is its neutral element, forming the algebraic monoid structure. Note that the purity of `mempty` implies that it always returns the same value, i.e., the neutral element is unique.

It is known from algebra, that lists are monoids. This can be expressed in Haskell with the following instantiation:

```
instance Monoid [b] where
  mempty  = []
  mappend = (++)
```

This states, that the binary operation is list concatenation (implemented by the infix operator ++) and the neutral element of list concatenation is the empty list.

Actually, this instantiation provides two additional interesting aspects. First, it is generic, as it defines an instance of `Monoid` for a list of any type, i.e., the type parameter `a` of the `Monoid` type class is instantiated with the type "list of values of type `b`" for any

b. And second, the definition of mappend is in so-called *point-free style*, i.e., it is defined in terms of another function without ever mentioning the actual function arguments. We will make use of lists as monoids in Section 5.1.3.2.

Basic type classes of Haskell are Show and Read to print and parse values, Eq and Ord to compare and sort values, Num, Real, Integral, etc. for numeric operations, Enum for enumerations, and Ix for array indexing. If a function wants to utilize the functions of a type class, it has to restrict its arguments to actually instantiate the respective type class. For instance, the function

```
sort :: Ord a => [a] -> [a]
```

can sort a list of values of type a, given a instantiates Ord, i.e., there actually is a total ordering for the values of the list.

Type classes are a distinguished feature of Haskell. Among other things, they enable the implementation of domain-specific embedded languages, as basic operations such as addition and comparison can easily be overloaded for new types. Type classes also play a major role in the implementation of monads, as the next section will explain.

2.3.2.3. Monads

Type classes can be parametrized not only by a type, but also by a type constructor. This means, the type parameter can be forced to be polymorphic. This is used by the Monad type class, which is defined as follows:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The function return wraps a value into the monad, yielding a *monadic value*. The infix operator >>=, sometimes called "bind", takes a monadic value, unwraps it, passes it to the given function, and returns its monadic result value. This allows for chaining functions that take pure values and return monadic values. We will provide an example further below.

Conceptually, a monad provides an encapsulation of side effects [141]. For example, failing can be regarded as a side effect, which leads to the Maybe monad:

```
instance Monad Maybe where
  return         = Just
  Nothing  >>= _ = Nothing
  (Just x) >>= f = f x
```

Lifting a value into the Maybe monad is simply wrapping it with the Just data constructor. Binding, on the other hand, distinguishes between two cases. If the input value already signals failure, then failure is returned without invoking the next function. To this end, the underscore binds the second argument to >>=, which effectively ignores that value. If there is an input value instead, >>= unwraps that value via pattern matching, passes it to the given function, and returns its result.

Given these functions, computations that might fail can be chained like this:

```
f :: a -> Maybe b
g :: b -> Maybe c
```

```
h :: c -> Maybe d

chain :: a -> Maybe d
chain x = f x >>= g >>= h >>= return
```

The implementation of chain aborts the pipeline with failure as soon as the first function fails and returns the final value otherwise. Consider the alternative of checking for failure after each single function via pattern matching.

```
chain :: a -> Maybe d
chain w = case f w of
            Nothing -> Nothing
            Just x  -> case g x of
              Nothing -> Nothing
              Just y  -> case h y
                Nothing -> Nothing
                Just z  -> Just z
```

The difference to the previous implementation shows that separating the concern of failure handling from the actual application logic leads to compact implementations. This is a great example of how higher-order functions (>>= in this case) support powerful and expressive abstractions.

Other major side effects are non-deterministic computations (provided by the list monad), stateful computations (provided by the reader, writer and state monads), and I/O operations (provided by the I/O monad). In either case, the signature of a function shows if it is pure or not, as in the latter case it returns a monadic value. The `Monad` type class enables the implementation of functions that can operate on any monad. For example,

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

is a monadic version of the `map` function that applies a function to each element of a given list and returns the resulting list. If `m` is a monad, then `mapM` performs the same computation while executing the side effects of `m`. For instance, if the provided function returns `Nothing` for one of the elements of the list, the overall computation yields `Nothing` as well, reflecting the semantics of the `Maybe` monad to fail the whole computation if a single interior step fails.

Particularly functions that return `IO b` for a given `b` are of interest, because they can be used to interact with the outside world [103]. While lazy evaluation itself does not guarantee the execution (i.e., the evaluation) of these functions in the right order, the I/O monad enforces it, ultimately enabling real world interactions in a purely functional environment. The type `IO` instantiates the `Monad` type class accordingly. The only special thing about that type is that there is no way to manually unwrap a value from the I/O context. Instead, each function that invokes a function that returns `IO` has to return `IO` itself, which is ultimately rooted in the `main` function, which has type `IO ()`. The type `()` is called *unit type* and its only value is *unit*, also depicted as `()`. A function which returns `IO ()` therefore declares statically that it returns no value but is only executed for its side effects. Overall, it is the type system which enforces that I/O can not be performed from within pure functions.

Haskell's monads became so ubiquitous, that a special syntax, the do-notation, has been introduced to ease their usage [140]. We will make use of this notation in Section 5.1.1. For instance, the previous example can also be written as

```
chain :: a -> Maybe d
chain v = do
  x <- f v
  y <- g x
  z <- h y
  return z
```

which expands to the previously given first definition of `chain`.

From this example, the benefit of the `do`-notation is not obvious. Therefore, consider the following example, which does not resemble a simple pipeline but uses intermediate results later on:

```
f,g,h :: Int -> Maybe Int

chain :: Int -> Maybe Int
chain v = do
  x <- f v
  y <- g v
  z <- h (x + y)
  return z
```

This example expands to

```
chain :: Int -> Maybe Int
chain v = f v
  >>= \x -> g v
  >>= \y -> h (x + y)
  >>= \z -> return z
```

where `\x -> expr` is a lambda expression, i.e., the definition of an anonymous function, which, in this case, takes a single argument and evaluates the given expression which might make use of that argument.

We want to close our explanation of monads with a final motivating example taken from practice. The Parsec library[5] provides an industrial-strength parser which is simple, safe, and fast. The major data type is `Parser`, the parser monad. It encapsulates the side effects of parsing, i.e., consuming input and back-tracking. The user can focus on the grammar itself to create the required parser.

For instance, the GDB manual [129] defines the following EBNF [63] for output records[6]:

```
output ->
  ( out-of-band-record )*
  [ result-record ]
  ( out-of-band-record )*
  "(gdb)" nl
```

Given parsers for out-of-band and result records, and a data type for outputs, the parser for output is as simple as this:

[5] `http://hackage.haskell.org/package/parsec`

[6] Actually, the documentation does not allow out-of-band records after the list of result records, but the current implementation behaves otherwise (bug ID 7708).

```
outOfBandRecord :: Parser OutOfBandRecord
resultRecord    :: Parser ResultRecord

data Output =
  Output [OutOfBandRecord] (Maybe ResultRecord)

output :: Parser Output
output = do
  oob  <- many        outOfBandRecord
  rr   <- optionMaybe resultRecord
  oob' <- many        outOfBandRecord
  string "(gdb)" >> newline >> eof
  return (Output (oob ++ oob') rr)
```

where `many` and `optionMaybe` are combinator functions which apply a given parser as often as possible, or return a `Maybe` value reflecting whether the given parser could be applied or not. Also, the infix operator >> is a variant of the bind operator which ignores the monadic value and evaluates the functions only for their side effects instead.

```
(>>) :: m a -> m b -> m b
x >> y = x >>= \_ -> y
```

So, the parser for outputs reads arbitrary many out-of-band records, optionally a result record, again arbitrary many out-of-band records, the string "(gdb) " followed by a newline and the end of the input. After parsing everything successfully, the parser returns an `Output` object which is constructed from the previously parsed components.

This example shows the power of monads, as the author of the parser only needs to understand how to use monads in general and trust the implementation to perform required side effects in the background. Also, the parser code is very close to the EBNF-based grammar definition. This is a good example of the expressiveness of Haskell.

2.4. Summary

In this chapter we established the context of our thesis. We first introduced necessary terminology by defining various terms concerning operating systems, applications, event-based and thread-based programming, preemptive and cooperative multitasking, as well as debugging. We also named common expectations software developers have from thread abstractions.

Subsequently, we presented the relevant state-of-the-art in event-based and thread-based programming, operating systems, thread libraries, compiler-assisted thread abstractions, and fault diagnostics tools. The focus has mostly been on the WSN domain, but we also discussed other relevant work.

Finally, we outlined the various approaches we have taken towards implementing our compiler prototype. We have ultimately chosen the Haskell programming language, which is why this chapter also introduces this language and a few of its core concepts.

3. Compiler: from T-code to E-code

In this chapter we investigate the primary goal of our work, the translation of thread-based applications (*T-code*) to equivalent event-based applications (*E-code*).

Section 3.1 provides an overview by sketching the translation scheme, listing its limitations (Section 3.1.1), and explaining its embedding into a surrounding project including interactions with various other tools involved (Section 3.1.2).

Section 3.2 explains the overall mapping performed by the translation scheme and covers its two aspects, data flow transformation (Section 3.2.1) and control flow transformation (Section 3.2.2). In Section 3.2.3 a small, but complete, example is provided.

Section 3.3 investigates translation semantics by specifying T-code semantics (Section 3.3.1), defining the term "equivalence" (Section 3.3.2), and by explaining why the translation scheme turns arbitrary but valid T-code into equivalent E-code (Section 3.3.3).

Section 3.4 covers the mediation between E-code and operating system (Section 3.4.1), and provides an implementation of a platform abstraction layer for Contiki (Section 3.4.2). It also shows proof of concept for TinyOS (Section 3.4.3), and discusses the generation of application-specific platform abstraction layers (Section 3.4.4).

Finally, Section 3.5 illustrates the compiler pipeline that performs the translation and covers its limitations, employed algorithms, and involved code representations.

3.1. System Overview

As depicted in Figure 3.1, in the context of our work a dedicated compiler translates a thread-based application (*T-code*) into an equivalent event-based application (*E-code*). The *T-code application* is built upon a synchronous *T-code API*, which has been manually derived from an asynchronous OS API. Instead of implementing the T-code API, what a run-time-based thread abstraction would do, the compiler additionally generates a corresponding *E-code API* that is used by the generated *E-code application*.

When translating the T-code API, the compiler does not alter auxiliary functions (cf. Section 2.1.4) and the operating system already provides an implementation for them. A blocking function is, in contrast, turned into a *yield point function*, i.e., an asynchronous function with a systematic signature and the same semantics as the corresponding blocking function. The *platform abstraction layer* (PAL) (Section 3.4) implements yield point functions by means of the OS API.

Regarding the application code, the compiler's task is to turn critical functions into event handlers (Section 3.2.2). Also, the compiler has to preserve the value of local variables across context switches if needed (Section 3.2.1). And the E-code application should be equivalent to the T-code application in order to ensure the behavior intended by the T-code developer (Section 3.3).

The compiler generates for each thread a static data structure containing all variables needed after a critical call. By replacing local variables with their static counterparts, the relevant state of the task stays available. Overall, the control flow transformation

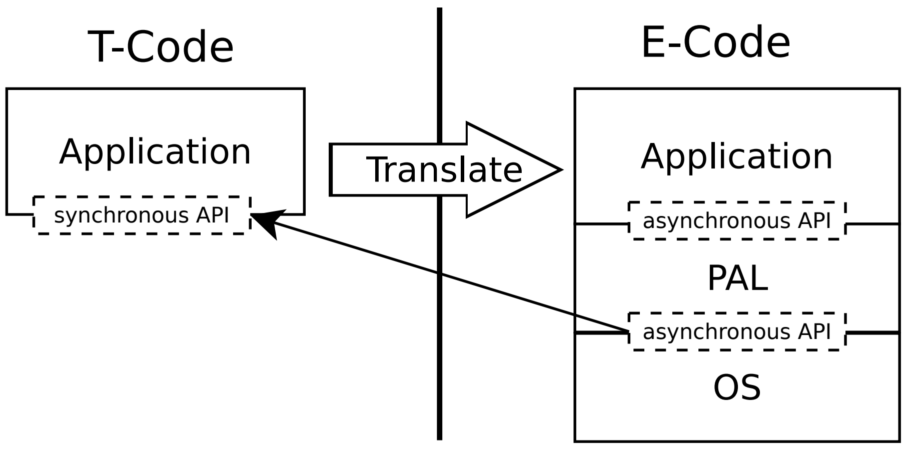


Figure 3.1.: *System overview: compiling:* The compiler translates the thread-based application (T-code) into an equivalent event-based application (E-code). The platform abstraction layer (PAL) mediates between the generated application and the given operating system (OS).

preserves statement execution order while the transformation of the data flow preserves the particular effects. This is why T-code and E-code are equivalent.

For T-code, a task is implemented by a thread and a set of critical functions. For E-code, on the other hand, the same task is implemented by a single function that contains the implementation of all involved critical functions. This function serves as the single event handler function and is called by the PAL to start or resume the task. Consequently, the task is suspended when the event handler function returns. This always happens immediately after the invocation of a yield point function. Multitasking is achieved by resuming a task while other tasks are suspended (Section 3.4.1).

The translation scheme has been designed to avoid the usage of *dynamic memory*, i.e., objects with allocated storage duration. This is common practice in embedded systems because with dynamic memory is it hard to guarantee that out-of-memory situations do not occur.

Both T-code and E-code are compliant with existing language standards. This particularly means that we, in contrast to other approaches, avoid new language keywords or unusual execution semantics. Our approach therefore enables the utilization of existing language tools and eases its adoption to practice (cf. Section 2.1.5).

3.1.1. Limitations

A major handicap of static analysis and translation is its limitation to decidable problems. In the context of compiler-assisted thread abstractions, this entails three restraints to enable a reliable determination of the application's call graph.

First, it is not allowed to take the address of a critical function. Case differentiation, which causes only moderate overhead, can easily be used instead. Second, critical functions may not be recursive. Recursion makes stack consumption estimation undecidable, which is why this technique is uncommon in embedded systems anyway. We therefore consider this restriction being minor as well.

And third, dynamic thread creation cannot be supported, but threads have to be assigned statically with thread start functions. To some extent, dynamic thread creation can be simulated with static thread creation by blocking a static thread immediately after start-up

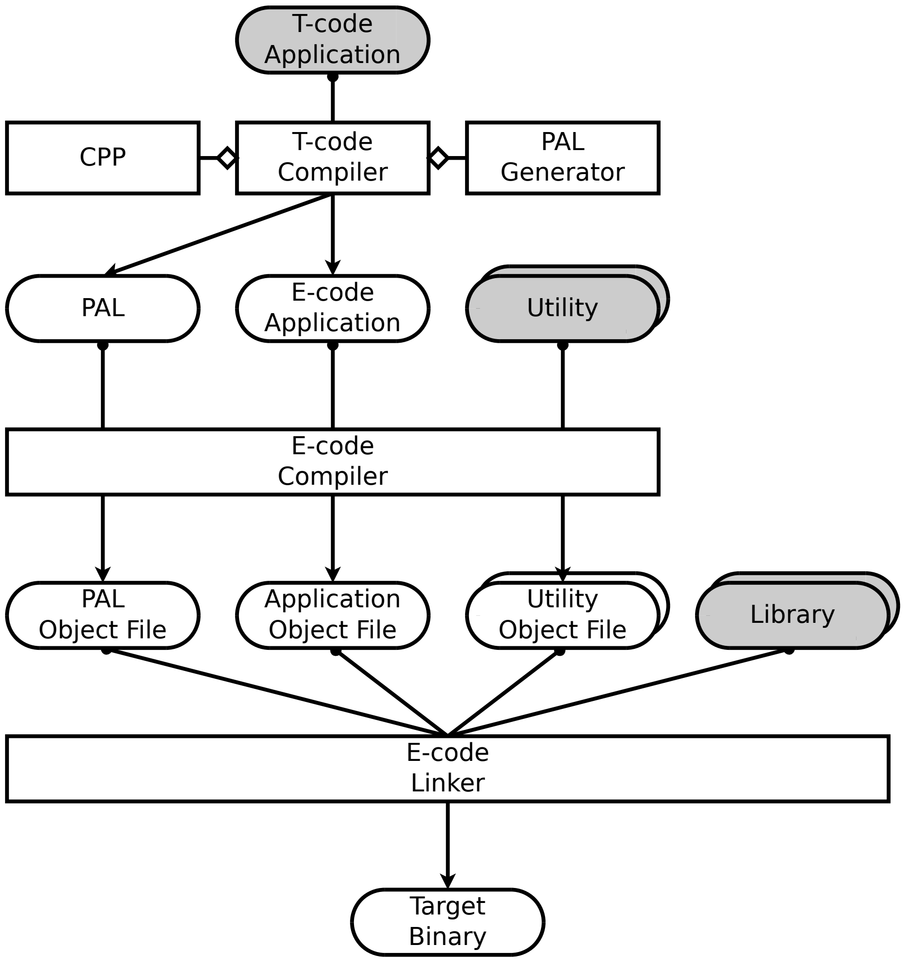


Figure 3.2.: *Project structure:* The T-code compiler turns T-code into E-code. The E-code tool chain is used to compile and link the E-code along with additional utility modules and libraries.

and releasing it when it is supposed to "start". This is a viable solution if the maximum number of "dynamically" started threads is known in advance. Since WSN applications are normally composed of a fixed set of tasks, this is not considered a severe limitation either.

These restrictions are all inherent to compiler-based approaches that avoid allocated storage, and the compiler reliably rejects invalid T-code. It therefore seems appropriate to refer to our approach as a comprehensive compiler-assisted thread abstraction.

3.1.2. Project Structure

Figure 3.2 shows the structure of a typical T-code project. A *T-code application* containing all critical functions and some auxiliary functions, as well as various utility modules and libraries constitute the input files of the project. The T-code application is translated into an *E-code application* by the *T-code compiler*. The utility modules and libraries are meanwhile processed by a conventional C tool chain. We use the prefix "E-code" to refer to this tool chain for clarification.

Other than an E-code compiler, a T-code compiler cannot process single *translation units* [C99: 5.1.1.1] independently from each other because it needs to determine the application's global call graph of critical functions. In principle, splitting the T-code application among multiple translation units is possible when passing them all to a single invocation of the compiler. However, merging translation units to obtain a global view is not expected to add insights regarding our research question. We have therefore chosen to require a single translation unit as input to the T-code compiler instead.

The T-code compiler depends on two external tools to perform its work: A C preprocessor to resolve preprocessor macros in the T-code application (Section 3.5.2), and a platform-dependent *PAL generator* to generate a platform abstraction layer that is optimized for the given application (Section 3.4.4). The output of the generator is standard C99 code and can be further processed by the E-code tool chain.

T-code can freely use any auxiliary function, whether it is defined in the T-code application, in a utility module, or in a library. The translation does not alter implementations nor invocations of auxiliary functions, therefore using the E-code tool chain to compile and link the E-code application along with the utility modules and libraries works seamlessly. The operating system in embedded systems usually comes as a set of libraries. The resulting target binary is self-sustaining and can be executed on a mote as is. Figure 3.2 illustrates such a setup, but the T-code compiler can also be used in different environments.

3.2. Translation Scheme

The translation's goal is to transform valid T-code into equivalent E-code. To this end, the translation distinguishes between critical and blocking functions, and everything else. All *external definitions* [C99: 6.9] concerning neither a critical nor a blocking function are passed through unchanged and are not considered any further. Declarations of blocking functions, in contrast, become declarations of yield point functions. And declarations of critical functions are discarded while their definitions are translated into an *intermediate representation* (IR) as a first step (Section 3.5.4).

Besides some technicalities such as uniqueness of identifiers, the IR requires two things in particular. First, advanced control flow structures like `while` and `for` loops are replaced by basic statements such as `if`, `goto`, and label statements. And second, critical calls have only two distinctive appearances. The *first normal form* is a statement consisting of a single critical call, and the *second normal form* is anything of the form

$$\underline{\texttt{expression}}\ \underline{\texttt{?=}}\ \underline{\texttt{function}}\texttt{(}\underline{\texttt{parameters}}\texttt{);}$$

where $\underline{\texttt{?=}}$ stands for any *assignment operator* [C99: 6.5.16].

To establish the normal form, the compiler makes use of the fact that critical calls in nested expressions can be substituted by new variables that are initialized by the same critical call in a directly preceding statement (Section 3.5.4.4). Special care is needed to handle *Boolean short-circuit evaluation* [C99: 6.5.13-4] [C99: 6.5.14-4] correctly, which is achieved by splitting Boolean expressions that contain critical calls into a sequence of `if` statements as needed (Section 3.5.4.3). Overall, the translation into IR preserves the effective control flow of the T-code application.

3.2.1. Data Flow

Given the IR, the compiler performs a liveness analysis on each critical function to find the set of *critical variables*, i.e., local variables whose value is needed after a critical

```
1  double c1(char* s){
2    int m = s + 1;
3    c2(m);
4    int i = f();
5    m = s + i;
6    c3(m);
7    return i;
8  }
```

(a) critical function

```
typedef struct {
  int m;
} eframe_c1_t;
```

(b) E-frame

```
typedef struct {          1
  void* cont;             2
  double result;          3
  char* s;                4
  int i;                  5
  union {                 6
    tframe_c2_t c2;       7
    tframe_c3_t c3;       8
  } frames;               9
} tframe_c1_t;            10
```

(c) T-frame

Figure 3.3.: *T-stack and E-frame of a critical function:* `c1`, `c2`, and `c3` are critical functions while `f` is an auxiliary function.

call such that it has to be preserved. Because aliasing turns liveness analysis into an undecidable problem, the compiler makes conservative choices such as considering a variable as critical if its address is taken somewhere (Section 3.5.4.7).

With the set of critical and non-critical variables at hand, the compiler generates a *T-frame* and an *E-frame* for each critical function. Figure 3.3a shows an example of a general critical function. The functions' E-frame (cf. Figure 3.3b) contains all of its non-critical variables. In Figure 3.3a, line 5 modifies the value of `m` before it is read in line 6. Its value, which is set in line 2 before the call to `c2`, is therefore not needed, which is why `m` is a non-critical variable. For each thread, the `union` of all E-frames of the involved critical functions establish the *E-stack* (Section 3.5.5.1), a local variable of the *thread execution function* as explained in the next section.

A T-frame (cf. Figure 3.3c) contains the *continuation*, i.e., information about where execution should continue when the critical function returns (line 2), the return value of the function if existent (line 3), its parameters (line 4), its critical variables (line 5), and a union of the T-frames of all critical callees (lines 6–9). Furthermore, one T-frame for each thread starting function, called the *T-stack* of the thread (Section 3.5.5.1), is instantiated statically. Function parameters, critical or not, are always stored on the T-stack to simplify the implementation of critical calls. This can be improved in the future.

Given these stacks, the compiler is rewriting access to local variables by replacing them with the corresponding variables (Section 3.5.5.2). Similarly, access to *function-static variables*, i.e., objects with static storage duration and with identifiers that have block scope, is rewritten because the single critical functions are dissolved. The following section investigates this.

3.2.2. Control Flow

To translate the control flow for each thread, the bodies of all respective critical functions are collected into one common *thread execution function*. This function serves as a single event handler function for all events of the corresponding task. While dissolving the functions, the compiler equips every first statement of a function body and every first statement after a critical call with a unique label, which serves as a continuation point. The compiler also translates all critical function calls and function returns (Section 3.5.5.2).

Figure 3.4 depicts an example of how critical calls to critical functions are replaced by the following sequence of statements: First, the callee's parameters are copied to the T-stack (line 4). Then, the continuation information for the callee is copied to the T-stack

```
__attribute__((tc_start)) void blinky() {
  while(1) {
      wait(23);
      // do something
  }
}
```

$\Downarrow$

```
 1  void thread_0(..) {
 2  //...
 3  blinky_1: if (!1) return;
 4    tstack_blinky.frames.wait.dt = 23;
 5    tstack_blinky.frames.wait.cont = &&blinky_2;
 6    goto wait_1;
 7  blinky_2:
 8    // do something
 9    goto blinky_1;
10
11  wait_1:
12    // E-code body of wait
13    goto *tstack_blinky.frames.wait.cont;
14  }
```

Figure 3.4.: *Critical call of a critical function:* `blinky` and `wait` are critical functions.

(line 5). Finally, a `goto` jumps to the start of the callee's body (line 6). Similarly, every `return` statement in T-code is replaced by a `goto` statement, which uses the continuation information stored on the T-stack (line 13). The replacement of the `while` loop by an `if` (line 3) and a `goto` statement (line 9) is, as previously stated, due to the translation into intermediate representation.

Figure 3.5 shows that critical calls to blocking functions are replaced by a slightly different sequence of statements. First, function parameters (line 5) and continuation information (line 6) are copied to the T-stack as well. But then, the yield point function is called, passing it a pointer to its T-frame (line 7). Lastly, the thread execution function returns in order to pass control back to the PAL (line 8). The PAL takes care to invoke the thread execution function `thread_0` again as soon as the operation has completed, passing it the continuation that has previously been copied to the T-stack of the yield point function. The first statement in each thread execution function is a `goto` (line 2) that resumes the computation at this location. The critical call now has a return value, i.e., the member `result` of the T-frame of `sleep`. If the critical call were in second normal form, the value would be copied from T-stack and assigned to the translated *lvalue* [C99: 6.3.2.1-1] of the normal form's assignment.

The compiler needs to determine the call graph of the application to be able to transform the control flow. This implies that it must be possible to identify thread start functions and blocking functions. As shown in Figure 3.4 and Figure 3.5, this is achieved by using *function attributes* [GNU C: 6.30] and defining new attribute names, i.e., `tc_start` and `tc_block`. The attribute extension has been carefully designed to minimize compatibility issues with other implementations. For example, it can easily be masked via a simple preprocessor macro that expands the pattern `__attribute__(x)` to nothing. Section 5.1.2.1 shows a variation of that technique.

The translation of the control flow employs *computed goto statements* which are also a GNU C extension [GNU C: 6.3]. In C99, it is not possible to take the address of a label and to use a pointer as the target of a `goto` statement. It is however possible to achieve the same effect by replacing the introductive `if` statement (cf. Figure 3.5, line 2) with a

```
__attribute__((tc_block)) _Bool sleep(int until);

void wait(int dt) {
  sleep(42 + dt);
  // do something
}
```

⇓

```
void thread_0(void* cont) {
    if (cont) goto *cont;
//... E-code body of blinky
wait_1:
  tstack_blinky.frames.wait.frames.sleep.until = 42 + tstack_blinky.frames.wait.dt;
  tstack_blinky.frames.wait.frames.sleep.cont = &&wait_2;
  sleep(&tstack_blinky.frames.wait.frames.sleep);
  return;
wait_2:
  // do something
  goto *tstack_blinky.frames.wait.cont;
}
```

Figure 3.5.: *Critical call of a blocking function:* `wait` is critical functions, and `sleep` is a blocking function.

`switch` statement whose body spans the whole function, and by turning all labels into `case` labels with different literal integers. Then, the continuation would be an integer instead of a pointer. Protothreads actually apply this technique.

3.2.3. Example

Figure 3.6 represents a small, but complete example. In both listings, the lines labeled with A are external definitions concerning neither a critical nor a blocking function. They are therefore passed unchanged from T-code to E-code.

Label B indicates the T-frame structures for the critical functions `sleep`, `wait`, and `blinky`. For example, the integer `state` (E-code, line 16) originates from the local variable `state` of the function `blinky` (T-code, line 9). Label C shows the instantiation of the T-stack.

Label D is the E-frame for the function `wait`, which is the only function with a non-critical variable. Empty T-frames are not generated, thus the E-stack (Label E) contains only one member.

Labels F and G indicate the bodies of the function `blinky` and `wait` with two modifications. First, access to variables and parameters is altered. For instance, access to `state` and `now` (T-code, line 10 and 20) is translated to access to the T-stack and E-stack (E-code, line 27 and 39). And second, the control flow is translated into continuation passing style. This involves instrumenting the code with labels marking the single continuations (E-code, line 28, 33, 38, and 44). It also involves rewriting critical calls to critical functions (E-code, line 30–32) and to blocking functions (E-code, line 40–43).

Label H shows how declarations of blocking functions become declarations of yield point functions. Also, line 6 and line 8 of the T-code show the syntax of function attributes that are used to mark thread start and blocking functions.

The actual implementation applies a naming scheme to new identifiers in order to guarantee uniqueness. For clarity this and other details are omitted in this example and we refer to the source distribution of our work [6] instead.

```
  | int delay = 500;
A | int get_leds() { /* ... */ }
  | void set_leds(int state) { /* ... */ }
  | int time(); // included from OS header file

H | __attribute__((tc_block)) _Bool sleep(int until);

    __attribute__((tc_thread)) void blinky() {
  |   unsigned char state;
  |   state = get_leds();
  |   while(1) {
F |     wait(delay);
  |     state ^= 0xff;
  |     set_leds(state);
  |   }
    }

    void wait(int dt) {
  |     int now;
G |     now = time();
  |     sleep(now + dt);
    }
```

⇓

```
  | int delay = 500;
A | int get_leds() { /* ... */ }
  | void set_leds(int state) { /* ... */ }
  | int time();

  | typedef struct {
  |   void* cont; _Bool result; int until;
  | } tframe_sleep_t;
  | typedef struct {
  |   void* cont;
  |   union { tframe_sleep_t sleep; } frames;
B |   int dt;
  | } tframe_wait_t;
  | typedef struct {
  |   union { tframe_wait_t wait; } frames;
  |   unsigned char state;
  | } tframe_blinky_t;

C | static tframe_blinky_t tstack_blinky;
D | typedef struct { int now; } eframe_wait_t;
H | void sleep(tframe_sleep_t*);

    void thread_0(void* cont) {
E |   union { eframe_wait_t wait; } estack;
      if (cont) goto *cont;

  |   tstack_blinky.state = get_leds();
  | blinky_1:
  |   if (!1) return;
  |   tstack_blinky.frames.wait.dt = delay;
  |   tstack_blinky.frames.wait.cont = &&blinky_2;
  |   goto wait_1;
F | blinky_2:
  |   tstack_blinky.state ^= 0xff;
  |   set_leds(tstack_blinky.state);
  |   goto blinky_1;

  | wait_1:
  |   estack.wait.now = time();
  |   tstack_blinky.frames.wait.frames.sleep.until = estack.wait.now + dt;
G |   tstack_blinky.frames.wait.frames.sleep.cont = &&wait_2;
  |   sleep(&tstack_blinky.frames.wait.frames.sleep);
  |   return;
  | wait_2;
  |   goto *tstack_blinky.frames.wait.cont;
    }
```

Figure 3.6.: *Translating T-code to E-code:* A small but complete example

3.3. Equivalence

It is important to ascertain that the generated code behaves as intended by the T-code developer. In this section it is therefore specified what it means for the E-code to be equivalent to the T-code.

As a first step, Section 3.3.1 defines the semantics of T-code, so that software developers know which behavior they can expect from their T-code application. In Section 3.3.2 the term "equivalence" itself is defined. Why the transformation from T-code to E-code preserves equivalence is discussed in Section 3.3.3.

The execution semantics of an E-code application is specified by C99. It is nondeterministic, because from a set of pending events either one could occur first. As we will see further on, the details of the underlying operating system are not important here as long as the PAL implements the generated E-code API properly (Section 3.4.1).

3.3.1. T-code Execution Semantics

A *T-code application* is a single C99 *preprocessing translation unit* [C99: 5.1.1.1] with GNU C extensions. The set of *valid* T-code applications, however, is a subset thereof for two reasons.

First, a T-code application must contain one or more definitions of thread start functions, and one declaration of each blocking function that is called by at least one critical function. It may also contain additional definitions of critical functions and other external definitions.

Second, there are a number of constraints for a valid T-code application. For example,

- critical functions must not be recursive,
- a *function designator* [C99: 6.3.2.1-4] must not reference a critical function,
- there must not be a *main function* [C99: 5.1.2.2.1],
- thread start functions must not take any parameters and must return `void`.
- the declaration of a blocking function must be annotated with the function attribute `tc_api`,
- the definition of a thread start function must be annotated with the function attribute `tc_thread`, and
- identifiers must not start with the prefix `ec_`.

A complete list of constraints that apply for the whole translation unit and for critical functions in particular can be found in Section 3.5.3.

The execution semantics of a T-code application is derived from C99 semantics. The execution semantics of a single T-code thread equals the execution semantics of a C99 application if its thread start function is replaced with a `void main()` function. From this basic principle, the execution semantics of the complete T-code application can be established as follows:

At any point in time, there is at most one thread in *running state*. Whenever a thread calls a blocking function it goes from running state to *blocking state*. If no thread is in running state, one of the blocking threads whose operation has completed in the meanwhile is selected nondeterministically. This thread then goes from blocking state to running state and resumes its execution as if the critical call that caused the thread to block previously had just returned. Switching context between threads is like suspending the associated C99 application of the hitherto running thread and resuming the associated C99 application of the next thread. Hereby, the following rules on the visibility of the application state apply.

Objects with static or allocated storage duration are all shared between the threads. This means that all threads read from and write to the same instance. In contrast, objects with automatic storage duration are thread-local. This means that there is one separate instance for each thread and each function call. Modifications of shared objects are guaranteed to be visible to other threads when resuming[1].

A thread is started automatically by calling its thread start function when the program starts. The threads of an application are started in source code order of their thread start functions. This rule is arbitrary, but also simple and intuitive. It is the duty of the PAL to enforce this rule (Section 3.4). A thread quits if its thread start function returns. If a thread calls `exit` [C99: 7.20.4.3] or its associated C99 application causes an *abnormal program termination* [C99: 5.1.2-1], the complete T-code application terminates immediately.

T-code is subject to the whole spectrum of *unspecified*, *undefined*, *implementation-defined*, and *local-specific behavior* of C99 applications [C99: 3.4] [C99: Annex J]. The *implementation* [C99: 3.12] in this case is a given combination of T-code and E-code compiler.

3.3.2. Equivalence Definition

It is important to ascertain that the generated code behaves as intended by the T-code developer. We therefore define an E-code application to be *equivalent* to a T-code application if and only if every possible *observable behavior* of the E-code corresponds to at least one possible observable behavior of the T-code. The execution semantics of both T-code and E-code applications are nondeterministic. Thus, a single program can have various observable behaviors.

The intuition behind this definition is twofold. If the observed interactions with the environment performed by the E-code application are indistinguishable from the interactions by the T-code application, then both applications apparently do "the same thing" and it does not matter which one is executed. And second, if every observed interaction of the E-code application can be explained by an execution of the T-code application, then "nothing surprising" can happen.

The *observable behavior* is defined as the actual run-time order of all invocations of API functions including the *values* of the involved function parameters. For a T-code application, this includes all calls of blocking functions and their function parameters. For an E-code application, it means all calls of yield point functions and the members of the corresponding T-frame that constitute the parameters of the blocking function. This definition is sound because yield point functions and T-frames are systematically generated from blocking functions.

The exact timing of function calls is not part of the observable behavior. Cooperative threads are not viable for timing-critical tasks anyway, such that this definition does not involve any additional restrictions. The implementation of yield point functions is not

[1]As our work targets single core motes this is trivially true.

covered either because it is not generated. Instead, it is the duty of the PAL author to ultimately preserve the semantics.

The *value* of a parameter is the "precise meaning of [its] contents" [C99: 3.17] if it has *basic*, *array*, *structure*, or *union type* [C99: 6.2.5]. If the parameter is of *pointer type* [C99: 6.2.5], its value is that of the referenced object. When comparing observable behavior programmatically, this definition makes it impossible to compare pointers to `void` because they cannot be dereferenced. Similarly, comparing arrays requires some way to determine the array size in order to compare the array elements. This can, for example, be achieved by knowing that the arrays are null-terminated or by having an additional length parameter at hand. Otherwise, comparing arrays is impossible as well. This, however, does not impede the soundness of our definition. The transformation preserves the observable behavior; in these cases we just cannot verify this fact at run-time.

3.3.3. Correctness of the Transformation

The previous sections have specified the definition of equivalence and the execution semantics of T-code and E-code applications. As a next step, the properties of the translation steps have to be elaborated in order to reason about their correctness.

The uncritical variables of a given critical function are stored on the E-stack. Being uncritical implies, by definition, that the value of a variable is not needed after a critical call. It does not matter that the E-frame shares memory with the E-frame of the callee, which is a critical function. It is also irrelevant that the E-stack is a local variable whose value is reset with every return of the corresponding thread execution function, because a return only happens after a critical call to a yield point function. Overall, E-stack variables can simulate the essential storage duration of corresponding T-code variables.

Critical variables of a given critical function are stored on the T-stack. They are local variables, which implies by definition that their values are not needed anymore when the corresponding function returns. It is therefore insignificant that the T-frame shares memory with T-frames of other critical functions. In fact, a T-stack is, by construction, the overlay of all snapshots of the hardware stack, given the corresponding thread would actually be executed.

Function parameters are stored on the T-stack whether critical or not. If the algorithm cannot decide if variables are critical, a conservative choice is made and they are stored on the T-stack as well. Storage duration on the T-stack outlasts storage duration on the E-stack, which is why T-stack variables can simulate the storage duration of corresponding T-code variables. Overall, data flow transformation preserves effects of function statements, as it only exchanges storage locations of involved variables.

Regarding control flow, we have to point out first, that translation into intermediate representation preserves the observable behavior of the T-code application (Section 3.5.4). The transition from threads to events is performed by the transformation of the control flow that operates on this representation. For any given thread, any execution path in T-code between two consecutive calls to blocking functions corresponds to exactly one invocation of the corresponding thread execution function in the E-code. As the control flow transformation preserves the sequence of statements while the data flow transformation preserves their effects, the observable behaviors of T-code and E-code are equivalent for each occurrence of an event. In order to show the equivalence of T-code and E-code in general, sequences of occurring events have to be compared next.

If there is only one task, the execution environment which causes varying results of yield point functions is the only source of nondeterminism for E-code. As yield point functions and corresponding blocking functions have the same semantics, the same nondeterminism also exists in T-code. Also, each sequence of events in E-code corresponds to a possible control flow in T-code, such that the only remaining concern is the interleaving of multiple tasks.

The execution of E-code is nondeterministic with regard to the order of occurring events. The event sequences of all executed tasks can interleave arbitrarily and one of all possible interleaved event sequences will happen at E-code run-time. However, the execution of T-code is nondeterministic as well because when two threads are blocked at the same time each one of them could continue next. Therefore, there is one possible control flow in T-code for each possible sequence of events in E-code. As the observable behavior of each step is equivalent, the overall equivalence can be deduced.

This reasoning is obviously informal. Finding a formal proof of equivalence is, however, very involved for various reasons. First, C99 provides no formal specification of execution semantics. Then, C99 includes unspecified, undefined, and implementation-defined behavior. And finally, the language as a whole is rather complex and involves many corner cases. To compensate for a lacking proof, we took a series of measures to verify the correctness in the case of the chosen case study applications of our evaluation (Section 6.2).

3.4. Platform Abstraction Layer

The purpose of the *platform abstraction layer* (PAL) is to provide an implementation for yield point functions and to drive the execution of application tasks. It is important to note that the PAL is not a conceptual requirement of compiler-assisted thread abstractions. Instead, the PAL connects existing operating systems with generated E-code and its complexity depends directly on how dissimilar a given OS API is from the systematic E-code API (Section 3.4.1). If a future operating system would provide that interface by itself, no PAL would be needed ultimately.

All of the existing operating systems, however, need some mitigation. The PAL can then also be used to integrate generated E-code with existing native code. It is even possible to integrate multiple E-code applications into a single executable, which allows development of independent T-code applications that can be executed jointly on a single mote. The PAL therefore not only introduces overhead but it also entails a lot of flexibility.

In Section 3.4.2 a minimalistic example of a PAL for Contiki is presented. Our evaluation in Chapter 6 employs a Contiki PAL built by following this approach. In Section 3.4.3 a proof of concept PAL for TinyOS PAL is provided. And finally, Section 3.4.4 explains how a specialized PAL for a given application can be generated to improve the overall efficiency.

3.4.1. E-code API

As mentioned in Section 3.1, the E-code API is generated from the T-code API, which was manually derived from the OS API. This does not imply that there is one blocking function for each asynchronous OS API function. Instead, arbitrary blocking functions with appropriate semantics can be specified as long as the OS API provides means to actually perform required operations. For instance, the example provided in Section 2.1.3, a `receive_with_timeout` function, could use the two OS API functions `listen` and `set_timeout`.

A yield point function that has been generated from a blocking function always returns `void` and takes a pointer to its T-frame as a single parameter. The T-frame contains all function parameters of the blocking function and an additional member for its result value. Additionally, the T-frame has a member that stores continuation information of the invoking task. The implementation of the yield point function arranges that the result is eventually copied to the T-frame if needed and the thread execution function of the invoking task is called with continuation information from the T-frame. Yield point functions are *strictly asynchronous*, which means that they must return immediately and they must ascertain that the thread execution function is called eventually. This is different to some implementations of asynchronous functions that return a failure and never invoke the registered callback if the requested operation could not be triggered. It is the duty of the PAL author to keep blocking function declarations (and documentation) and yield point function implementations synchronized with respect to function semantics.

The PAL must start tasks in the source code order of the corresponding thread start functions. To start a task, the PAL calls the thread execution function passing it a `NULL` continuation. After that, the task will call a yield point function which will take appropriate actions and return. Subsequently, the thread execution function returns as well, passing control back to the PAL. The requested operation could either not be triggered or it will complete eventually. In any case, when the thread execution function is called the result of the operation has been copied to the T-stack and the task resumes from the point of continuation that is passed to the thread execution function.

Spurious events that emerge in the operating system without being triggered by a call to a yield point function must be ignored instead of being propagated to the application level. This maintains thread semantics, which specify that a thread can only react to happenings in the environment while being blocked in a call to a blocking function. In other words, a thread cannot resume without being blocked previously. The PAL has to ensure this.

The PAL drives the execution of all tasks and has to keep an account of which task called which blocking function. If a thread execution function returns without invoking a yield point function in advance, the corresponding task must quit, i.e., that thread execution function must not be invoked any more. Also, while one task is waiting for the requested operation to complete, the PAL can call thread execution functions of other tasks and resume them. This is the essential of the multitasking E-code application.

3.4.2. Contiki

To illustrate the building blocks of a Contiki PAL, we assume the minimalistic T-code example employing only a single thread and the single blocking function `sleep` from Figure 3.6. As already depicted there, the translation generates the T-frame `tframe_sleep_t` and the declaration of the yield point function `sleep`. The PAL needs to implement this function and drive a proper execution of `thread_0`, the thread execution function of the single thread.

Contiki is built upon an event-based approach with a single event handler function for each task. Usually, a task is implemented by a protothread, which provides the syntactical illusion of blocking functions. Nevertheless, Contiki is an event-based system at run-time. For clarity, we avoided using any of the protothread macros for the Contiki PAL. We are still using a so-called “process”. This is unavoidable, but without protothreads a process

```
 1  typedef enum {
 2    YPF_none = 0,
 3    YPF_sleep,
 4    /* constants for other yield point functions*/
 5  } YPF;
 6
 7  typedef struct {
 8    union {
 9        struct {
10            frame_sleep_t* frame;
11            struct etimer et;
12        } sleep;
13        /* contexts of other yield point functions*/
14    } context;
15    YPF ypf;
16  } ThreadContext;
17
18  ThreadContext threads[1];
19  ThreadContext* current_thread;
20
21  void sleep(tframe_sleep_t* frame) {
22    current_thread->context.sleep.frame = frame;
23    current_thread->ypf = YPF_sleep;
24    clock_time_t now = clock_time();
25    if (frame->until > now) {
26      etimer_set(&thread->context.sleep.et, frame->until - now);
27    } else {
28      process_post(PROCESS_CURRENT(), PROCESS_EVENT_CONTINUE, NULL);
29    }
30  }
```

Figure 3.7.: *Contiki PAL: implementation of* `sleep`

is nothing more than an event handler plus meta information. In order to use Contiki's event multiplexing mechanism it is advisable to employ one process per T-code thread.

Figure 3.7 shows the implementation of the yield point function `sleep`. The array `threads` stores one `ThreadContext` for each T-code thread (line 18). A `ThreadContext` remembers which yield point function has been called (line 15) and saves associated context, which is the T-frame (line 10) including additional state necessary for the implementation (line 11). As each thread can call no more than one yield point function at any point in time, the contexts can share memory via a union (line 8–14). The pointer `current_thread` (line 19) always points to the `ThreadContext` instance of the thread that is currently running.

The implementation of `sleep` first saves the T-frame (line 22) and the selection of the yield point function (line 23) to the context of the current thread. Next, it queries the current time from an auxiliary function of the operating system (line 24). If the requested time lies in the future (line 25), the timer is set (line 26), causing a timer event being sent to the current process once the time expires. If instead the requested time has already elapsed, a continuation event is sent to the current process immediately (line 28). This implements strict asynchronism of the yield point function.

Figure 3.8 lists the event handler function for the Contiki process that drives the T-code thread. Whenever this function is invoked, the current thread pointer is set properly (line 3). The function can discern its first invocation by looking at (line 5) and updating (line 6) meta information in `ptinfo`. If the function is invoked for the first time, the `continuation` is set to `NULL` (line 7). As line 26 calls the thread execution function with `continuation`, this effectively starts the thread. Otherwise, by looking at the context of the current thread (line 10), the handler code for the current yield point function is

```
 1  static char event_handler_0(struct pt* ptinfo, process_event event, process_data_t data) {
 2    void* cont;
 3    current_thread = &threads[0];
 4
 5    if (ptinfo->lc == 0) { // first invokation
 6      ptinfo->lc = 1;
 7      cont = NULL;
 8    }
 9
10    else if (current_thread->ypf == YPF_sleep) {
11      if (event == PROCESS_EVENT_TIMER) {
12        current_thread->context.sleep.frame->result = true;
13      } else { // event == PROCESS_EVENT_CONTINUE
14        current_thread->context.sleep.frame->result = false;
15      }
16      continuation = current_thread->context.sleep.frame->cont;
17    }
18
19    /* handle other yield point functions*/
20
21    else { // spurious event
22      return PT_YIELDED;
23    }
24
25    current_thread->ypf = YPF_none;
26    thread_0(continuation);
27    return PT_YIELDED;
28  }
```

Figure 3.8.: *Contiki PAL: event handler running a T-code thread*

executed. In this example, there is only the `sleep` function. If the current event is a timer event (line 11), the result of the yield point function, which is stored on the T-stack, is set to `true` (line 12). Otherwise, the current event must be a continuation event, in which case the result is set to `false` (line 14). In any case, `continuation` is set to the caller's continuation, which is stored on the T-stack (line 16). Line 26 again invokes the thread execution function, resuming the thread in this case. Whenever the tread execution function is called, it will eventually call a yield point function, which will take proper action, as shown previously, and return. The thread execution function will then return too, passing control back to line 27 of the event handler function. This concludes the event handler function by signaling Contiki that further events for the current process are expected. Spurious events are ignored (line 21–23, line 25). The same mechanism additionally implements task termination.

Figures 3.7 and 3.8 sketch how to extend the PAL to support more yield point functions and more T-code threads. For the former, the enumeration `YPF` and the union `ThreadContext.context` have to be extended, and the yield point functions have to be implemented. For additional T-code threads, extra Contiki processes are needed, each using a different instance from the `threads` array, a different thread execution function, and a different event handler function. To avoid code duplication, the common event handler code should then be moved to a shared function that only operates on `current_thread` (Section 5.1.2.2).

As the PAL is just one of many Contiki modules, it can and does use standard Contiki primitives to communicate with others. Integrating existing native code poses therefore no problems.

```
 1  interface Pal {
 2    command void sleep(tframe_sleep_t* frame);
 3    /* commands for other yield point functions */
 4
 5    command void toggle_led_0();
 6    command uint32_t time();
 7    /* more auxiliary functions if required */
 8
 9    // the thread execution function
10    event void thread(void* continuation);
11  }
```

Figure 3.9.: *TinyOS PAL: the PAL interface:* The incarnation of the E-code API.

3.4.3. TinyOS

PALs for other operating systems look different, but in general it should always be possible to perform the necessary mapping. Figure 3.9 shows the interface definition of the E-code API for a particular thread. There we can see that yield point functions are implemented as TinyOS commands (line 2) and the thread execution function is implemented as a TinyOS event (line 10). The interface additionally contains arbitrary auxiliary functions, also implemented as commands (line 5–6).

Figure 3.10 shows a TinyOS component that implements the PAL interface (line 11) for a particular thread. There is an enumeration (line 16–20) and a selector (line 27) for the current yield point function. A union stores the context of each of these functions (line 22–25), and a variable holds the current continuation (line 28). When the yield point function `sleep` is called (line 40) and the requested time lies in the future, the selection is memorized (line 43), the T-frame is stored (line 44), and a timer is triggered (line 45). When the requested time has already passed, `continuation` is set to the caller's continuation (line 47) and the thread execution function is invoked.

In contrast to Contiki, invoking the thread execution function is decoupled with a scheduler task being posted (line 37, 48, and 55), whose implementation signals the event for the thread execution function with the current `continuation` (line 30). When the system starts, `continuation` is set to `NULL` (line 36) before posting the scheduler task, thus starting the thread. Likewise, when an anticipated event occurs, the caller's continuation is copied from the T-frame to `continuation` before posting the scheduler task (line 54). Again, spurious events are ignored (line 31, 53) while the same mechanism implements task termination.

Because TinyOS uses nesC and the T-code compiler generates C code, an additional step is required to unify them. One possible way is to use the skeleton module shown in Figure 3.11, add the T-frame declarations at the top and implement the `thread` event handler with the body of the generated thread execution function. To achieve this, calls to both E-code API functions and auxiliary functions have to be converted from C-style to nesC style by adding the `call` keyword. Also, the current T-code compiler makes use of computed `goto` statements which is a GNU extension that is not supported by nesC. The T-code compiler therefore has to be extended with a compatibility mode using a central `switch` statement and constants instead of computed `goto` and label statements (cf. Section 3.2.2).

Finally, these three components have to be wired together, adding further triples for additional threads, and wiring everything into a complete TinyOS application. We have performed this process manually and wrote some basic unit tests verifying that this

```
module ThreadC @safe()
{
  uses interface Boot;

  uses interface Timer<TMilli>;
  /* dependencies of other yield point functions */

  uses interface Leds;
  /* dependencies of other auxiliary functions */

  provides interface Pal;
}

implementation
{
  typedef enum {
    YPF_none = 0,
    YPF_sleep,
    /* constants for other yield point functions */
  } YPF;

  union {
      tframe_sleep_t* sleep;
      /* contexts of other yield point functions */
  } context;

  YPF ypf;
  void* continuation;

  task void scheduler() {
    ypf = YPF_none;
    signal Pal.thread(continuation);
  }

  event void Boot.booted() {
    continuation = NULL;
    post scheduler();
  }

  command void Pal.sleep(tframe_sleep_t* frame) {
    uint32_t now = call Timer.getNow();
    if (frame->until > now) {
      ypf = YPF_sleep;
      context.sleep = frame;
      call Timer.startOneShot(frame->until - now);
    } else {
      continuation = frame->cont;
      post scheduler();
    }
  }

  event void Timer.fired() {
    if (ypf == YPF_sleep) {
      continuation = context.sleep->continuation;
      post scheduler();
    }
  }

  command void Pal.toggle_led_0() {
    call Leds.led0Toggle();
  }

  command uint32_t Pal.time() {
    return call Timer.getNow();
  }
}
```

Figure 3.10.: *TinyOS PAL: the thread component:* This executes a single T-code thread.

```
// T-frame declarations

module UserlandC {
  uses interface Pal;
}

implementation {
  tframe_blinky_t tstack_blinky;

  event void Pal.thread(void* cont) {
    // implementation of the thread execution function goes here
  }
}
```

Figure 3.11.: *TinyOS PAL: the thread execution function:* A skeleton module for the implementation of the thread execution function.

process yields a functional PAL for an exemplary TinyOS application. This serves only as a proof of concept.

3.4.4. PAL Generator

Because the PAL introduces a resource-wise overhead to the E-code application, it is important to keep it as small as possible. An obvious way to do so is having a PAL specialized to the application's needs instead of a full-fledged PAL supporting all possible applications. This suggests to employ a *PAL generator* that takes the application's properties and generates a tailored PAL.

The application's properties, referred to as the *application footprint*, comprise the number of threads and the set of utilized yield point functions per thread. Using this information, the PAL generator for Contiki, for instance, can create a constant size `threads` array. Also, only if a particular yield point functions is in use, its implementation, event handler code and thread context is actually included in the PAL. The details of the PAL generator interface and our Contiki implementation used in the evaluation are covered in Section 5.1.2.2. PAL generators for other operating systems should be able to perform similar application-specific optimizations.

3.5. Compiler Pipeline

In this section, we will cover the distinct stages of the compiler pipeline, which is depicted in Figure 3.12.

The input to the compiler is a T-code application, which is a translation unit in *concrete syntax representation* (CSR) [C99: Annex A]. The Parser stage (Section 3.5.2) determines the *abstract syntax representation* (ASR) of this translation unit, which is an *abstract syntax tree* (AST) (Appendix A). The Analysis stage (Section 3.5.3) computes the static call graph of the T-code application and checks its validity with respect to a set of constraints. The Front-end stage (Section 3.5.4) turns the ASR of each critical function into an *intermediate representation* (IR) (Section 3.5.1) and performs additional validity checks. The Back-end stage (Section 3.5.5) takes the IR of all critical functions and performs the translation from threads to events as described in Section 3.2. This results in the ASR of the E-code application, which the Printer stage (Section 3.5.6) turns to CSR.

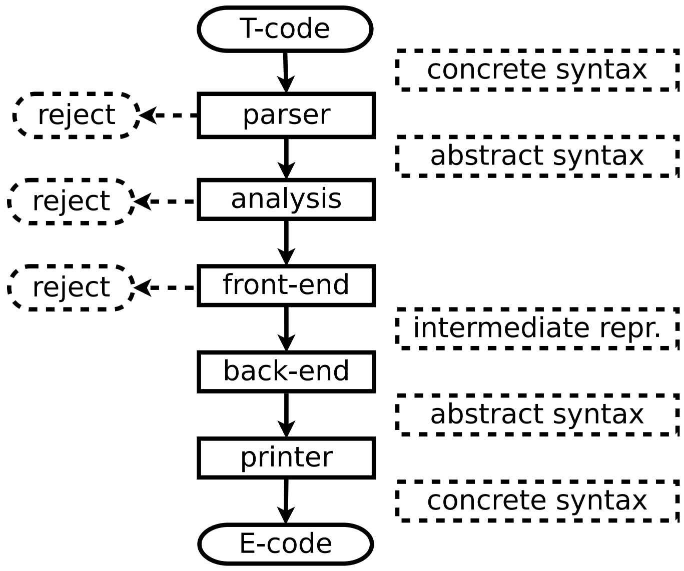


Figure 3.12.: *The compiler pipeline:* Five consecutive stages turn valid T-code applications into equivalent E-code applications.

While the design of the compiler pipeline is not special by itself, its internals involve many concepts that are particular for translating threads to events. The following sections will therefore explain the functionality of each stage. To provide an abstract illustration of the concepts, we describe the stages as a sequence of pseudo functions operating on pseudo data types. While their names serve as a reference to the implementation, their signatures are considerably simplified so that we can focus on concepts rather than implementation.

Because we claim that our T-code compiler either rejects T-code as invalid or turns it into equivalent E-code, we have to emphasize the aspect of completeness. The following sections will therefore provide a complete specification for what constitutes valid T-code. Additionally, as the C programming language is complex and knows many corner cases, they will carefully keep track of what the input and output domains of the various functions are in order to argue why the whole spectrum of valid T-code applications is covered overall.

Instead of inventing our own notation, we will use (almost) valid Haskell [4] syntax for the specifications of the pseudo functions and pseudo data types. In most cases, the syntax uses intuitive symbols like parenthesis to group tuples and square brackets to denote lists (cf. Section 2.3.2). We therefore consider it an adequate tool for the intended purpose. In any case, we will explain the syntax along the way.

As the execution semantics of T-code is derived from C99 (cf. Section 3.3.1), T-code is subject to all kinds of unspecified, undefined, implementation-defined and locale-specific behavior. Thus, it is advised to first compile a T-code application using the E-code compiler (cf. Section 3.1.2) with warnings enabled. At first, processing T-code with the E-code compiler might sound wrong, but T-code is valid C99 code (with GNU C extensions). Also, the E-code compiler is the one that is used to generate the target binaries for the given platform, so its choice of behavior [C99: J.3-1] is what matters ultimately. The resulting object file is not used, as the purpose of the compilation is to check for behavioral issues only. In the following, we assume that such issues have been addressed properly.

3.5.1. Intermediate Representation

The purpose of the *intermediate representation* (IR) is to both facilitate data flow analysis and optimizations and to simplify the implementation of the back-end by reducing the number of possible cases. Integrating critical calls into the IR makes this representation unique in comparison to standard intermediate representations.

Just as T-code, the IR represents thread-based C code in a way which is independent of the actual translation scheme. In contrast to T-code, however, only critical functions can be expressed in IR, as everything else passes the compiler unmodified. The top-level IR data type is therefore a function, which consists of variables and a function body.

Data 3.1: `Function` =
```
[Variable]  --  scoped variables
Body        --  function body
```

A `Variable` represents an identifier with block scope that refers to an object with automatic or static storage duration, i.e., a local or function-static variable that has been declared within the corresponding function. It contains the original name of the identifier, its declaration with a unique name, and some additional flags. These flags specify the storage duration of the variable, whether it is a function parameter or not, whether it is critical or uncritical, and whether it originates from the T-code application or has been introduced by a transformation step of the compiler pipeline.

Data 3.2: `Variable` =
```
String  --  original name
CDecl   --  renamed declaration
[Flag]  --  flags
```

A `Body` consists on an *entry label* and a list of basic blocks, each of which is associated with a unique name to which we refer to as *IR label*.

Data 3.3: `Body` =
```
Label                   --  entry label
[(Label, BasicBlock)]   --  labeled basic blocks
```

A basic block consists of exactly one *entry node*, zero or more *middle nodes*, and exactly one *exit node*.

Data 3.4: `BasicBlock` =
```
EntryNode     --  exactly one entry node
[MiddleNode]  --  zero or more middle nodes
ExitNode      --  exactly one exit node
```

Entry nodes allow control to flow in, middle nodes always pass control through and exit nodes either terminate the control flow or redirect it to other basic blocks. Each node particularly contains a statement, and the execution semantics of a node corresponds to the execution semantics of that statement. Table 3.1 lists the possible IR nodes with their contained statement.

There are two kind of entry nodes. The simpler one is just a *label node* which contains a label statement whose attached statement must be a *null statement* [C99: 6.8.3-3]. Thus, executing a label node has no effect. If a basic block starts with a label node, its IR label is equal to the identifier of the label statement of that label node.

The second kind of entry node is a *continuation node* which contains a critical call

node type	**node name**	**contained statement**
entry node	label node	`label: ;`
	continuation node	`expression ?= function(parameters);`
middle node	expression node	`expression;`
exit node	`return` node	`return expression;` `return;`
	`goto` node	`goto target;`
	`if` node	`if (condition) goto target1; else goto target2;`
	critical call node	`function(parameters);` `expression ?= function(parameters);`

Table 3.1.: *IR nodes and their contained statements*

in second normal form and represents the execution that takes place when that critical call returns. As defined in Section 3.2, a critical call in second normal form assigns the value of the function call to an arbitrary *modifiable lvalue* [C99: 6.3.2.1-1]. Thus, when executing a continuation node, this assignment is actually performed. If a basic block starts with a continuation label, its IR label is assigned to a unique name.

Concerning middle nodes, there are only *expression nodes* that contain a single *expression statement* [C99: 6.8.3]. That statement must not comprise a critical call. When executing an expression node, the contained expression statement "is evaluated as a void expression for its side effects" [C99: 6.8.3-2]. Non-critical function calls are evaluated just like other expressions for their side effects, ignoring the fact that the control actually leaves the function in between.

Finally, there are four kinds of exit nodes.

1. A *return node* contains a `return` statement and terminates the control flow. The `return` statement may carry a return expression which must not comprise a critical call.

2. A *goto node* contains a `goto` statement and redirects the control flow to exactly one basic block that must start with a label node.

3. An *if node* contains an `if` statement and redirects the control flow to one of two basic blocks that must start with a label node. This implies that the contained `if` statement must be in the *else form* [C99: 6.8.4.1-2] and both of its sub-statements must be `goto` statements. Executing an `if` node has the same behavior as executing the contained `if` statement, i.e., the selection of the subsequent basic block depends on the condition of the `if` statement.

4. A *critical call node* contains a critical call in first or second normal form. If a critical call node is executed then the contained function call is performed for its side effect. The fact that the control flow actually leaves the function in between and also might block is ignored. A critical call node also carries the label of the basic block that resumes execution. If the critical call is in first normal form then the targeted block must start with a label node. If the critical call is in second normal form instead, then the targeted block must start with a continuation node.

The *entry label* specifies the first basic block. The control flow starts with the entry node of that block and always terminates at a `return` node.

3.5.2. Parser

The Parser is the first stage of the compiler pipeline. It turns a translation unit from CSR to ASR. The most interesting aspect of this stage is that a T-code application actually is a *preprocessing translation unit* [C99: 5.1.1.1], i.e., it may contain *preprocessing directives* [C99: 6.10]. The abstract syntax, however, cannot represent those. Instead, they have to be *preprocessed* [C99: 6.10.1-6], yielding the CSR of the actual translation unit. This step can, among other things, produce additional critical calls that are not obvious in the preprocessing translation unit, which is another reason why it has to be done foremost.

We refer to the output of this step as *P-code*, which is an abbreviation for "preprocessed T-code". In this terminology, the translation from threads to events is actually a translation from P-code to E-code. We will, however, still say "T-code to E-code translation" when the distinction is not important.

Function 3.1: `preprocess ::`
```
    Path       --  path to file containing the input T-code
                   application
 →  String     --  a command line for the execution of the C
                   preprocessor
 →  Either     --  either
      [Error]      --  a list of errors, or
      P-Code       --  the output from the preprocessor
```

The specification of a pseudo function uses the symbol `::` to separate the function name from the parameter list, which is a list of data types separated by the → symbol. The last parameter in the list is the return type of the function. So, the `preprocess` function takes both, the path to a file as well as a command line string, and returns either a list of errors or the corresponding P-code.

As preprocessing is highly dependent on the environment, the user has to provide an external tool to actually generate the P-code. The given command line specifies how to invoke that tool (Section 5.1.2.1). It is an error if the input file cannot be opened or read, if the preprocessor cannot be executed, or if the preprocessor fails to generate output.

If no error occurs, the P-code can be parsed next, yielding the ASR of the T-code application. We use a parser implementation from a third party, i.e., the Language.C library [57], which supports K&R C [66], C99 [64] and several GNU extensions [GNU C: 6].

Function 3.2: `parse ::`
```
    P-Code              --  a single translation unit in CSR
                            without preprocessing directives
 →  Either              --  either
      [Error]               --  a list of parse errors, or
      CTranslationUnit      --  the P-code in ASR
```

3.5.3. Analysis

The Analysis stage is the second stage of the compiler pipeline. It determines the static call graph of the input application and applies a set of constraints to it.

As our compiler guarantees that if a T-code application is accepted then the resulting E-code application is equivalent, it has to filter T-code applications for which this guarantee cannot be given. This concerns two types of features. First, the set of valid T-code applications is a subset of C99 with GNU extensions due to conceptual limitations (cf. Section 3.1.1 and Section 3.3.1). And second, our prototypical implementation does in fact not support the complete set of valid T-code applications, mainly in order to stay focused on the research question of this work. At this stage, the filtering concerns a few features that are rejected globally. Later, additional constraints will be enforced for critical functions.

Function 3.3: `global_constraints ::`

```
    CTranslationUnit  --  the AST of a translation unit with
                          K&R C features, C99 features, and GNU
                          extensions
 →  [Error]           --  a possibly empty list of reasons why
                          the AST has been rejected
```

The `global_constraints` function checks the AST for compliance with what the compiler can process. It is an error if the input AST violates any of the following restrictions:

- limitations on C99:
  - **reserved prefix**: The T-code compiler must be able to generate unique identifiers. The easiest way to do so is to use a proper naming scheme together with a reserved prefix. The current implementation uses the prefix `ec_`, which thus must not be used by a T-code application.
  - **main function**: T-code threads are started via thread execution functions only. We thus prohibit the definition of a `main` function to avoid confusion.
- excluded GNU extensions:
  - **nested functions:** We do not support function definitions inside of function definitions [GNU C: 6.4], because this gives rise to scoping and re-used identifiers, which would complicate the compiler implementation.
  - **thread-local storage**: GNU C supports thread-local storage which “is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread” [GNU C: 6.59]. In this context, “thread” refers to a preemptive thread like those provided by pthreads (IEEE Std 1003.1c-1995). Thread-local storage is a useful feature, but it can effectively be provided by a set of auxiliary API functions in combination with *cast operators* [C99: 6.5.4] with minor overhead. Thus, we have chosen to not include direct support for this feature in our compiler, although we expect doing so to be straight forward.

Now that in particular nested function definitions are excluded, the compiler can determine the static call graph of the application. The call graph is an important source of information for various subsequent steps.

Function 3.4: `call_graph ::`
```
    CTranslationUnit  --  a T-code application in ASR, in
                          particular without nested function
                          definitions
 →  CallGraph         --  the static call graph of the
                          application
```

The `call_graph` function determines the static *call graph* of the translation unit by scanning for function calls while ignoring calls via function pointer. It then identifies the set of blocking functions and determines each reachable sub-graph. Each function from the union of those is finally flagged as a critical function. Note that this includes functions that call a blocking function but are not called by a thread execution function. This is needed to check if the address of such a function is taken somewhere, thus enabling a critical call path that is not detected by the static analysis (Section 3.5.4.8).

With the knowledge of critical functions, additional constraints can be enforced on them. As already mentioned, some of the constraints are conceptual limitations and others are meant to keep the set of special cases in the later stages of the compiler manageable for the scope of this work. Overall, we believe that the set of supported features still allows for reasonable development of C applications, and our evaluation in Chapter 6 supports this.

Function 3.5: `critical_constraints ::`
```
    CTranslationUnit  --  the T-code application in ASR
 →  CallGraph         --  the static call graph
 →  [Error]           --  a possibly empty list of reasons why
                          the AST has been rejected
```

There are two groups of C99 features that are not supported for critical functions. The first group consists of conceptual limitations and consequences derived from them:

- **critical recursion:** Recursion of critical functions is not allowed.
- **no thread start function:** For obvious reasons, T-code applications that do not start a single thread cannot be transformed in a meaningful way. Thus, the compiler rejects such cases.
- **thread not yielding:** If a cooperative thread never calls a blocking function it will run forever and prevent other threads from being executed. This can hardly be intentional, which is why the compiler rejects thread start functions that are not critical. Note that the former property is a syntactic attribute while the latter one is derived from the factual call graph. Also note that a critical thread start function still might never call a blocking function at run-time, for example because it is trapped in an endless loop that does not contain a critical call. The question if a thread will eventually block at run-time is undecidable, which is why static analysis cannot give a definitive answer to that matter.
- **signature of thread start function:** A thread start function is never called, which

is why it makes no sense if it returns a value or takes parameters. To emphasize this, the compiler enforces the following function signature: `void function()`.

- **parameter names:** Parameters in function declarations are not forced to have a name. In case of blocking functions this poses a problem because the implementation of the PAL needs a handle to access the right data from the T-frame. We thus enforce parameter names for the declaration of blocking functions.

The second group concerns features that we think are conceptually possible to support, but which are disallowed in critical functions to avoid the associated complexity in the compiler. Also, we expect little insight regarding the research question of this work from supporting these features. The extend of the following list reflects the complexity of the C programming language. Because the T-code compiler is expected to reliably distinguish between valid and invalid input, we have to consider all of these cases.

- **critical group of declarators:** In C99, a *declaration* [C99: 6.7] can contain zero or more arbitrary *declarators* [C99: 6.7.5]. If the *function declarator* [C99: 6.7.5.3] of a critical function is not the only declarator of the enclosing declaration, the input is rejected. An example would be "`int i, c(int k), *j;`", given `c` is a critical function. As due to our experience, function declarators are rarely mixed with other declarators in practice, we think that this restriction is reasonable.

- **volatile type qualifier:** In C99, "an object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects" [C99: 6.7.3-6]. This is needed to implement drivers that access memory-mapped hardware registers, for instance. Accessing a volatile object is a *side effect* that has to be completed after each *sequence point* [C99: 5.1.2.3]. However, the T-code to E-code transformation does not necessarily preserve sequence points, which means that a T-code application cannot rely on the semantics of volatile objects. Thus, the compiler rejects T-code applications that make use of them. In principle, the transformation could be carefully designed to preserve the order of sequence points, but regarding the fact that our work targets application and service layer components that usually do not depend on this feature, avoiding the additional complexity in the compiler seems to be reasonable.

- **switch statements:** The C99 syntax is very flexible when it comes to `switch` statements. To facilitate the generation of the IR, we enforce the following constraints:
  - The body of the `switch` statement is a compound block.
  - The first statement of this compound block is a `case` statement or a `default` statement.
  - There is at most one `default` statement. (Actually, C99 requires this, but the parser does not enforce it.)
  - There are no `case` statements after a `default` statement.

  Each of these restrictions actually represents good practise, so we consider them being acceptable.

- **inline assembler:** We do not support inlined assembler code because the transformation would have to consider it, thus becoming platform-dependent. In principle, though, assembler code can be transformed accordingly (cf. Section 6.5).

- **ellipses:** With ellipses, the structure of the T-stacks would become very complex, and the macros from `stdarg.h` [C99: 7.15] such as `va_start` would have to be redefined. Ellipses are a core feature of the C programming language, so a future version of the compiler should support them.
- **array initializer lists:** In C99, one can enrich a declaration of a variable of array, structure or union type with an *initializer lists* [C99: 6.7.8] which provides initial values. When separating declaration and initialization, which is what the transformation does, one can resort to *compound literals* [C99: 6.5.2.5] using the same initializer list. However, this does not work for arrays, because C does not support array assignment. Instead, array compound literals have the same semantics as variables of array type, i.e., they are converted to a pointer to the first array element. Possible workarounds are to wrap the array in a structure which then can be assigned, to use `memcpy` [C99: 7.21.2.1] to copy the arrays manually, or to change the type of the variable from `T[]` to `T*` and to assign the pointers. It is currently unclear to us, which of these approaches, if any, provides a general and robust solution to this problem. Thus, we cannot support this case in good conscience. We still support initializer list for structure or union types.

While the parser supports K&R C and GNU C extensions, supporting these dialects adds a lot of complexity to the compiler. Thus, we disallow to use them within critical functions[2]. While the following list of excluded features limits what is supported for critical functions, at the same time it specifies what is possible for auxiliary functions. The list is also complete in that sense that any additional features are not supported by the parser. For each of these features we think that supporting them is conceptually possible.

- excluded K&R features:
  - **implicit return type:** K&R C allows that function return types implicitly default to **`int`**.
  - **K&R-style function declarations:** K&R C allows to decouple the declaration of the types of function parameters from the function declaration. One can even omit the type declaration completely in which case it defaults to **`int`**.
- excluded GNU extensions:
  - **compound statements as expressions:** In C99, a statement can be an expression and an expression is a recursive structure consisting of expressions. In GNU C, an expression can additionally contain a *compound statement* [C99: 6.8.2], thus creating a loop in the type system [GNU C: 6.1].
  - **computed gotos:** In C99, the target of a `goto` statement has to be a label. GNU C additionally supports to take the address of a label, store it in a `void` pointer and use this value for a *computed goto statement* [GNU C: 6.3]. Note that although the compiler does not support computed gotos, the generated E-code makes use of them nevertheless.
  - **array range designators:** GNU C supports to address ranges of array fields [GNU C: 6.26].

[2]To be precise, omitting the middle operant of a *conditional expression* [C99: 6.5.15] is a GNU C extension [GNU C: 6.7]. This feature is supported for critical functions nevertheless.

- **case ranges:** In C99, a `case` label must be an integer constant expression [C99: 6.8.4.2-3]. GNU C additionally supports the specification of value ranges for case labels [GNU C: 6.27].
- **attributes**: GNU C supports *function attributes* "which help the compiler optimize function calls and check your code more carefully" [GNU C: 6.30]. T-code makes use of this feature to identify blocking functions and thread start functions. But the T-code compiler does not implement the semantics of the various predefined attributes. Additionally, attributes for variable declarations [GNU C: 6.36] and type declarations [GNU C: 6.37] are rejected as well. We consider some of them, such as the `packed` attribute, which "specifies that each member [...] of the [attributed] structure or union is placed to minimize the memory required" [GNU C: 6.37], rather useful. A future version of the T-code compiler should therefore support them after carefully checking their implications on the equivalence of T-code and E-code.
- **built-ins:** The compiler does not support the following GNU built-ins:
  - `__builtin_offsetof` [GNU C: 6.50]
  - `__builtin_va_arg` [GNU C: 6.5]
  - `__builtin_types_compatible_p` [GNU C: 6.54]

The reader might have noticed, that `critical_constraints` does not check if the address of a critical function is taken somewhere, although Section 3.1.1 lists this as one of the conceptual constraints. In fact, this analysis is not possible without tracking nested scopes, because a new identifier might happen to have the same name as a critical function, in which case applying the *address operator* [C99: 6.5.3.2] to this identifier would misleadingly trigger an error. Nested scopes are dealt with in the Front-end, thus this analysis is deferred until then.

3.5.4. Front-end

The Front-end is the third stage of the compiler pipeline. It determines the IR of each critical function and enforces some final constraints on them.

First, `collect` and `desugar` simplify the function body by reducing the set of employed features. Then, `short_circuit` and `normalize` establish normal form for critical calls. The resulting code is a semantics-preserving rewrite of the input code and constitutes a T-code application that could actually have been written by a software developer.

Next, `basic_blocks` turns the ASR of a critical function into the corresponding IR by determining its basic blocks. Subsequently, `optimize` and `critical_variables` perform basic optimizations and static analysis on the IR. Finally, `filter` enforces some final constraints on critical functions.

Most of these functions use the call graph that has been determined by the Analysis module. We omit such environmental information from the function signatures for clarity.

3.5.4.1. collect

The first step is to separate declarations from statements. Under certain circumstances, which will be described below, it is possible to move all declarations to the beginning of a function without modifying the semantics of its implementation. This is what `collect` does in principle, only that the declarations are in fact removed from the function body and returned separately.

Function 3.6: `collect ::`
```
    CFunctionDef      --  the definition of a critical
                          function in ASR
 →  (                 --  a tuple consisting of
       [CBlockItem],      --  the body of the function as a list
                              of ASR block items containing only
                              statements and no declarations, and
       [Variable]         --  the list of extracted variable
                              declarations in IR
    )
```

There are two possible places where declarations can occur: as a block item, or as the first component of a `for` loop. Both of them can be included into a compound statement of arbitrary nesting depth, so the algorithm has to recursively walk of the complete function body. If a declaration is augmented with an *initializer* [C99: 6.7.8], it is replaced with an assignment to the declared object using the initializer's value. Obviously, this is always possible for *basic types*, *pointer types*, and *function types* (the various types are defined in [C99: 6.2.5]). Due to compound literals, assigning initializer values is also possible for *structure types* and *union types*.

Because `collect` detaches variables from their surrounding scope, it assigns unique names to them to avoid confusion. These new names do not matter to the T-code developer, because the E-code debugger hides such details. We therefore do not go into the details here.

3.5.4.2. desugar

The next step is to "desugar" advanced control flow structures, i.e., substitute them with basic control flow structures while preserving the semantics of the program. This is what the `desugar` function does. Additionally, it flattens compound statements by replacing them with their contained statements. The result is a list of basic statements without nested scopes.

Function 3.7: `desugar ::`
```
    [CBlockItem]      --  the body of a critical function as
                          list of block items containing only
                          statements
 →  (                 --  a tuple consisting of
       [CStatement],      --  the body of the input function as a
                              list of basic statements, and
       [Variable]         --  the list of new IR variables
    )
```

A compound statement consists of a list of block items, and a block item can in principle be a statement or a declaration. At this point in the compiler pipeline, all block items are in fact statements because `collect` removed all declarations. Thus, by replacing compound statements with their list of statements, we not only flatten the code but also

statement	pattern
expression	`expression;`
`goto`	`goto label;`
`if`	`if (condition) goto label1;`
`else` form	`if (condition) goto label1; else goto label2;`
label	`label: ;`
null statement	`;`
`return void`	`return;`
`return`	`return expression;`

Table 3.2.: *Basic statements*

overcome the need for the generic block item type. This is why `collect` returns a list of statements instead.

The set of *basic statements* is summarized in Table 3.2. The given patterns are strict, which means that, for instance, label statements with other statements than null statements are not allowed. Table 3.3 lists how all control flow structures are substituted with basic statements. The substitutions are of course applied recursively and the new variables and new labels have unique names. Within a substitution pattern, the `body` of a statement is a possibly empty list of arbitrary statements. Also, the remarks indicate how `break` and `continue` statements within such a body are substituted. We can verify via Appendix A that these substitutions indeed cover all possibilities.

A few additional comments on two of the patterns are advisable. First, when substituting `for` loops, each of the three parts of the loop header may be missing. In each case, the corresponding output statement is omitted as well. Second, when substituting `switch` statements, the number of `case` statements can be anything greater or equal to zero. In any case, each `case` statement is simply substituted by the given pattern resulting in the equal amount of `if` statements and labelled bodies. Additionally, the `default` statement may be missing, in which case the trailing `goto` statement and the corresponding labelled body are omitted. Note that at this point in the pipeline possible `for` and `switch` statements are limited to what is covered by the given input patterns.

3.5.4.3. short_circuit

C99 specifies that "if the first operand [of a logical and-expression] compares equal to 0, the second operand is not evaluated" [C99: 6.5.13-4]. Likewise, "if the first operand [of a logical or-expression] compares unequal to 0, the second operand is not evaluated" [C99: 6.5.14-4]. This behavior is usually referred to as *short-circuit evaluation*.

Because `normalize` will replace critical calls with new variables that are initialized by that critical call right before their usage, the short-circuit evaluation will effectively not be performed. Thus, the compiler emulates this behavior in these cases. `short_circuit` substitutes Boolean expressions that contain critical calls with a sequence of statements that preserve the observable behavior. This list contains an `if` statement that skips the evaluation of the right-hand side when indicated.

Figure 3.13 shows two examples which reveal that double negation is used to get the Boolean value of an arbitrary expression independent of its type. They also show that all the new statements are in fact basic statements, as this is what is expected in the compiler pipeline at this point. And of course, unique names are assigned to new variables and new

statement	input pattern	output	remarks

label	`name: statement;`	`name: ;`
		`statement;`	

if	`if (condition) {`
	`  body`
	`}`
		`if (condition)`
		`  goto label1;`
		`  else goto label2;`
		`label1: ;`
		`body`
		`label2: ;`	

if-else	`if (condition) {`
	`  body1`
	`} else {`
	`  body2`
	`}`
		`if (condition)`
		`  goto label1;`
		`  else goto label2;`
		`label1: ;`
		`body1`
		`goto label3;`
		`label2: ;`
		`body2`
		`label3: ;`	

while	`while (condition) {`
	`  body`
	`}`
		`label1: ;`
		`if (!(condition))`
		`  goto label2;`
		`body`
		`goto label1;`
		`label2: ;`
			`break`
			⇒ `goto label2;`
			`continue`
			⇒ `goto label1;`

do	`do {`
	`  body`
	`} while(condition}`
		`label1: ;`
		`body`
		`if (condition)`
		`  goto label1;`
		`label2: ;`
			`break`
			⇒ `goto label2;`
			`continue`
			⇒ `goto label1;`

for	`for (init; condition; incr) {`
	`  body`
	`}`
		`init;`
		`label1: ;`
		`if (!(condition))`
		`  goto label3;`
		`body`
		`label2: ;`
		`incr;`
		`goto label1;`
		`label3: ;`
			`break`
			⇒ `goto label3;`
			`continue`
			⇒ `goto label2;`

switch	`switch (expression){`
	`  case const1: body1`
	`  default: body2`
	`}`
		`int var1 = expression;`
		`if (var1 == const1)`
		`  goto label1;`
		`goto label3;`
		`label1: ;`
		`body1`
		`label2: ;`
		`body2`
		`label3: ;`
			`break`
			⇒ `goto label3;`

Table 3.3.: *Substitution of control flow structures*

```
x = c() || y;
```

⇒

```
var1 = !!c();
if (! var1) goto label1;
var1 = !!y;
label1: ;
x = var1;
```

(a) base case

```
return ((i = x, e) && c() + 1) || y;
```

⇒

```
var1 = !!(i = x, e);
if (var1) goto label1;
var1 = !!(c() + 1);
label1: ;
var2 = !!var1;
if (!var2) goto label2;
var2 = !!h;
label2: ;
return var2;
```

(b) generic case

Figure 3.13.: *Explicit Boolean short-circuit evaluation:* `c` is a critical function.

labels.

Function 3.8: `short_circuit ::`

```
    [CStatement]      --  the body of a critical function as a
                          list of basic statement
 →  (                 --  a tuple consisting of
      [CStatement],       --  the body of the input function
                              without Boolean expressions that
                              involve critical calls, and
      [Variable]          --  a list of new IR variables
    )
```

The algorithms of `short_circuit` are a bit involved which is why we refer to Appendix B for its details.

3.5.4.4. normalize

As function calls can appear arbitrarily nested in any expression, this step brings critical calls into normal form to facilitate the actual transformation from threads to events. This reduces the amount of possible cases int the Back-end, thus facilitating the T-code to E-code transformation.

Overall, `normalize` scans each input statement in turn for a critical call that is not in normal form. Each of them is replaced by a new variable with a unique name and a preceding statement that assigns the value of the critical call to the new variable. This assignment now constitutes a critical call in second normal form Figure 3.14 shows two examples.

```
return c(i);
```

⇒

```
var1 = c(i);
return var1;
```

(a) basic case

```
j = c(23) + c(42);
```

⇒

```
var1 = c(23);
var2  = c(42);
j = var1 + var2;
```

(b) two critical calls

Figure 3.14.: *Normalization of critical calls:* `c` is a critical function.

Function 3.9: `normalize ::`

```
    [CStatement]     --  the body of a critical function as
                         a list of basic statements without
                         critical calls in logical operations
 →  (                --  a tuple consisting of
       [CStatement],     --  the body of the input function as
                             a list of basic statements and with
                             critical calls only appearing in
                             normal form, and
       [Variable]        --  a list of new IR variables
    )
```

The type of the new variable is the return type of the critical function that is called. Figure 3.14b shows in particular that if an expression contains multiple critical calls the generated assignments are in source code order of the corresponding critical calls. This — just like any other choice — is compliant with C99, as "the order of evaluation of subexpressions and the order in which side effects take place are both unspecified" [C99: 6.5-3].

3.5.4.5. basic_blocks

At this stage of the compiler pipeline, everything is prepared for the final translation into the intermediate representation. This is done by the `basic_blocks` function.

Function 3.10: `basic_blocks ::`

```
    [CStatement]  --  a function body as a list of basic
                      statements with critical calls in normal
                      form
 →  Body          --  the function body in IR
```

As explained in Section 3.5.1, an IR body is an entry label and a list of labeled basic blocks, while a basic block consist of a single entry node, zero or more middle nodes and a single exit node. To perform this transformation, `basic_blocks` performs the following steps:

1. null statements are removed from the input
2. each basic statement is annotated with the corresponding IR node type

annotation	basic statement
entry node	label
middle node	expression

exit node	`goto`
	`if`
	`else` form
	`return void`
	`return`

Table 3.4.: *Relation between basic statements and IR node annotations:* We refer to Table 3.2 for the list of basic statements.

3. the list of annotated statements is partitioned into prototypical basic blocks (i.e., a *proto-block*)
4. each proto-block is converted into a real basic block

We will address the non-trivial steps 2–4 in turn.

annotating Table 3.4 shows which basic statements are annotated with which IR node type. When annotating a statement this indicates that this statement is going to become an IR node of the annotated node type (cf. Section 3.5.1). However, the statement may still have the wrong structure. For example, a basic `if` statement is not required to be in `else` form, but an IR `if` node does. Such situations will be resolved by the conversion step.

partition A proto-block already has an IR label but contains annotated basic statements instead of nodes. More precisely, a proto-block consists of arbitrary many statements of type middle node and optionally one statement of exit node type. This implies that a proto-block might have no statements at all.

The list of proto-blocks are created by scanning the list of input statements and by splitting them the right positions as follows: If the first statement is a label statement then its identifier is used as the IR label of a new proto-block. Otherwise, a unique IR label is associated with the new proto-block. All subsequent statements of middle node type are added to the current proto-block. If the next statement is of type exit node it is used to finalize the current proto-block. If it is of type entry node instead, the current proto-block is left without a finalizing statement. In any case, the algorithm is applied recursively to the rest of the input statements resulting in additional proto-blocks.

If the end of the list of statements is reached while consuming middle nodes, then the input function has one or more code paths that are not terminated by a `return` statement. This is in general valid, because "a function may have any number of return statements" [C99: 6.8.6.4-2]. But at the same time, "if the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined" [C99: 6.9.1-12]. The second premise of this rule is excluded for functions returning `void`. For any other function, it is in general undecidable if the closing bracket will actually be reached at run-time. Thus, it is the burden of the developer to avoid the undefined behavior either by finishing all code paths with return statements or by not using the value of the function call when indicated.

Consequently, if the end of the list of input statements is reached while consuming middle nodes, the last block is finished with an explicit `return` node that returns `void`.

This is the correct behavior for functions returning `void` and for situations in which the undefined behavior is not triggered. If it is, the algorithm still behaves standard compliant, as "possible undefined behavior ranges from ignoring the situation completely with unpredictable results, [...]" [C99: 3.4.3.-2]. The same situation arises if the last proto-block is finished with a critical call or an `if` statement that is not in `else` form, as in these cases the control flow also reaches the closing bracket. Using the same rationale, an additional proto-block containing a return statement that returns `void` is appended to the list of proto-blocks in such a case.

convert The conversion turns proto-blocks into basic blocks using the following steps:

- The IR label of the basic blocks equals the IR label of the proto-block.
- An entry node is created as described below.
- All statements with type middle node are carried over and wrapped in middle nodes.
- If the proto-block is finished by a statement of exit node type its structure is completed if necessary and wrapped in an exit node. Otherwise, a new exit node is created as described below.
- The IR label of the first proto-block becomes the entry label of the returned `Body`.

Converting proto-blocks to basic blocks requires context information about the previous and the next proto-block in the sequence. If the previous proto-block was not finished by a critical call in second normal form, the IR label of the proto-block is turned into a label statement that is wrapped by a label node. Otherwise, the previous critical call is wrapped into a continuation node. If a proto-block has no statement of type exit node, a `goto` node is appended using the IR label of the next proto-block as the target. If there is a statement of type exit node it might be an `if` statement that is not in `else` form. In this case a new goto statement becomes its new second sub-statement, thus turning it into `else` form.

3.5.4.6. optimize

Given the IR, the compiler can now apply optimizations to it.

Function 3.11: `optimize ::`
```
    Body  --  the body of a critical function in IR
  → Body  --  the possibly improved body of the input function
```

The current implementation only applies a very basic optimization, which is the removal of minimal blocks. A *minimal block* is a basic block that consists of a label node and a `goto` node only, thus its only effect is the redirection of the control flow to another basic block. Thus, a minimal block can be removed if all places that target this minimal block are rewritten to target the basic block that is targeted by the minimal block instead. This potentially concerns all exit nodes as well as the entry label. This optimization reduces the amount of basic blocks and removes unnecessary redirections.

Of course, one can think of additional and more advanced optimizations such as constant folding and dead code elimination. In fact, the IR has been designed to facilitate the usage of Hoopl [108], a rich third-party Haskell library. Hoopl provides an implementation of the Lerner-Grove-Chambers algorithm for interleaved analysis and transformation

[75], thus enabling powerful optimizations that are composed of separate analysis and translation steps. We primarily make use of this library in the next stage of the compiler pipeline, but the present stage is the right place for additional optimizations.

3.5.4.7. critical_variables

Until now, a lot of effort has been put into converting the input program from its abstract syntax to its intermediate representation. Now it is time to capitalize on this.

Function 3.12: `critical_variables ::`
```
    Function  --  a critical function in IR
 → Function  --  the input function with variables flagged as
                 critical or non-critical
```

`critical_variables` determines the set of *non-critical variables* of a critical function. A *non-critical variable* is a local variable for which there is no code path from a critical function call to a read access of that variable. All the other local variables are referred to as *critical variables*. The distinction is important, because only critical variables have to be persisted during a critical call (cf. Section 3.2.1).

The variables of the input function are initially all flagged as critical, because this is a safe default. In contrast, the variables of the output function are flagged as non-critical if appropriate. To this end, the compiler performs a liveness analysis on the function body by providing a proper data flow lattice and a proper transfer function to Hoopl [108]. The result is a set of live identifiers for each basic block. From these basic blocks, the algorithm takes those that are targeted by at least one critical call node and unifies their sets of live identifiers. These identifiers also include objects with file scope, objects with static storage duration, functions, etc. Thus, the algorithm needs to filter by intersecting with the input list of variables.

The algorithm assumes that each read access to a variable was preceded by a write access. This conforms to C99, because “if an object that has automatic storage duration is not initialized explicitly, its value is indeterminate” [C99: 6.7.8-10] and the behavior is undefined if “the value of an object with automatic storage duration is used while it is indeterminate” [C99: J.2].

The liveness analysis fails to recognize read and write access to storage locations via pointers, as this problem is in general undecidable. Thus, as a safe default, the algorithm marks a variable as critical if its address is taken somewhere. Similarly, a variable of array type is also marked as critical, because its members can be accessed via pointers.

3.5.4.8. filter

Calling functions via function pointers prevents a static determination of the factual call graph. The translation, however, depends on knowing the call graph of critical functions. The compiler therefore rejects calls to critical functions via pointers.

The easiest way to do so is to prevent the address of a function to be taken in the first place. With the IR at hand, which has no nested scopes and whose identifiers are guaranteed to be unique, enforcing this rule is finally possible.

Function 3.13: `filter ::`
```
    Function  --  a critical function in IR
 → [Error]    --  a possible empty list of reasons why the input
                  function has been rejected
```

Conceptually, `filter` can enforce arbitrary constraints on the IR of critical functions. The only thing that is enforced currently is that it is prohibited to take the address of a critical function. First, it seems to suffice to scan for address operators whose operant is the identifier of a critical function. However, "a function designator [i.e., an expression that has function type] with type 'function returning type' is converted to an expression that has type 'pointer to function returning type'" [C99: 6.3.2.1-4]. This means, that the address of a function can be obtained without using the address operator, which makes the filter algorithm more involved.

It recursively scans the expressions that are contained in the statement of the IR nodes and checks all identifiers. If an identifier of a critical or blocking function is not directly enclosed in a function call node (`CCall`), it constitutes a function designator and is therefore reported as an error.

3.5.5. Back-end

The Back-end is the fourth stage of the compiler pipeline. It performs the actual translation from threads to events according to the scheme described in Section 3.2. As we will discuss in Section 6.5, that scheme is only one of many possible ones. The previous steps of the compiler pipeline have been designed to be generic in the sense that the generated IR assumes no knowledge from the Back-end.

Formally, the Back-end turns the set of critical functions in IR into a single translation unit in ASR. To this end, it employs three functions, namely `tstacks`, `estacks`, and `threads`, which generate the T-stacks and -frames, the E-stacks and -frames, and the tread execution functions. The Back-end additionally turns declarations of blocking functions into declarations of yield point functions, and IR variables for function-static variables into variable declarations with file scope, local linkage and a unique identifier.

Given these components and the input AST, the Back-end composes two results. First, it generates the *PAL footprint* which is a list of declarations of yield point functions along with the declarations of their T-frames, encompassing only functions that are actually used by the T-code application. This list is passed to the PAL generator along with the application footprint (cf. Section 3.4.4). Second, the Back-end generates the actual E-code application in ASR, which consists of

- the external declarations of the T-code application that do not concern critical or blocking functions,
- all T-frame declarations,
- all T-stack definitions,
- all E-frame declarations,
- all declarations of yield point functions,
- all definitions of translated function-static variables, and

- all thread execution function definitions.

3.5.5.1. Frames and Stacks

The function `estacks` generates the E-frames and E-stacks of the E-code application. It makes use of the call graph that has been determined by the Analysis stage. We again omit such environmental information from the function signatures in this section for clarity.

Function 3.14: `estacks ::`
```
    [Function]          --  the list of all critical functions
                            in IR
 →  (                   --  a tuple consisting of
       [CDeclaration],      --  the list of E-frames, and
       [CDeclaration]       --  the list of E-stacks
    )
```

As explained in Section 3.2, the E-stack contains the union of all E-frames of the critical functions that are involved in a thread. And for a given critical function, the E-frame contains the declarations of all non-critical variables that are not function parameters (cf. Figure 3.3b). This covers both variables that originate from the T-code application and variables that have been introduced by the Front-end.

All members of an E-frame have unique names within the scope of that frame. Thus collecting them into a single structure and addressing them from the thread execution functions is possible. Likewise, all E-frames have unique names as well. If an E-frame is empty, it is completely omitted. Similarly, if an E-stack is empty, it is also completely omitted.

The function `tstacks` generates the T-stacks and T-frames of the E-code application. As explained in Section 3.2, there is one T-stack per thread and one T-frame for each critical function and yield point function.

Function 3.15: `tstacks ::`
```
    [CDeclaration]  --  the list of blocking function
                        declarations in ASR
 →  [Function]      --  the list of critical functions in IR
 →  [CDeclaration]  --  one T-frame per critical function or
                        yield point function, and one T-stack
                        per thread execution function, each in
                        ASR
```

All members of a T-frame have unique names within the scope of that frame. It is therefore possible to collect them in a structure and to address them from the thread execution functions. Likewise, a T-stack is a variable with file scope, local linkage, and a unique name.

3.5.5.2. Thread Execution Functions

Finally, everything is prepared to actually perform the translation from threads to events. The `threads` function generates the thread execution functions of the E-code application (cf. Section 3.2.2).

node	**basic statement**	**statement**
label	`label: ;`	`label: ;`
continuation	`expression ?= function(parameters);`	`expression ?= tstack.tframe.result;`
expression	`expression;`	`expression;`
`goto`	`goto target;`	`goto target;`
`if`	`if (condition) goto target1; else goto target2;`	`if (condition) goto target1; else goto target2;`
`return`		
void	`return;`	`goto *tstack.tframe.cont;`
expression	`return expression;`	`tstack.tframe.result = expression; goto *tstack.tframe.cont;`
TSF	`return;`	`return;`
critical call		
CF	`function(e1, e2);`	`tstack.tframe.p1 = e1; tstack.tframe.p2 = e2; tstack.tframe.cont = &&label; goto function_label;`
YPF	`function(e1, e2);`	`tstack.tframe.p1 = e1; tstack.tframe.p2 = e2; tstack.tframe.cont = &&label; function(&tstack.tframe); return;`

Table 3.5.: *Expansion of IR nodes to language statements:* We refer to Table 3.1 for a list of IR nodes. “TSF” means “thread start function”, “CF” means critical function, and “YPF” means “yield point function”.

Function 3.16: `threads ::`
```
    [Function]      --  the list of critical functions in IR
  → [CDeclaration]  --  zero or one E-stack per thread
  → [CFunctionDef]  --  one thread execution function per thread
```

A thread execution function is a function definition with external linkage whose name is unique, reflects its source code order, and is well-known to the PAL. Embedding a critical function into a thread execution function implies translating its IR to ASR. This comes down to translating basic blocks and nodes to statements. The first basic block in control flow order is the one named by the entry label of the function. The subsequent basic blocks follow in a depth-first order of the control flow. However, basic blocks that start with a label node and are only targeted by a single exit node are excluded from this sequence. Instead, these basic blocks are inlined as described further below.

When translating a basic block, its nodes are expanded in order. As listed in Table 3.5, a label node is expanded into a label statement using the IR label of the basic block as the label’s identifier. In case of a continuation node, an expression statement is generated that assigns the return value of the previously called critical function to the left-hand side of the critical call. The return value is hereby copied from the T-stack. Middle nodes are simply unwrapped.

Ignoring inlining of basic blocks for now, both `goto` and `if` nodes are also simply unwrapped. In contrast, a `return` node yields a computed `goto` statement that resumes execution as indicated by the caller's continuation that is stored in the T-frame. If the basic `return` statement that is contained in the IR node has a return expression, then this return expression is assigned to the result variable of the T-frame in advance. In case of a thread start function, a `return` node yields a single `return` statement instead. This makes the thread execution function return without previously invoking a yield point function, thus terminating the thread.

A critical call node is expanded into a sequence of statements that perform the critical call as explained in Section 3.2.2. This involves assigning values to the callee's function parameters, saving the continuation address, and performing a `goto` to the callee's entry label. In case of a yield point function, a function call is performed instead, followed by a `return`.

Inlining a basic block first involves expanding its middle nodes and its exit node to a list of statements as described above. This list then replaces the `goto` statement that would otherwise be expanded from the `goto` or `if` node that targets that basic block. This simple optimization reduces the code size and avoids unnecessary redirections at run-time.

While expanding IR nodes, expressions are scanned for variables that have to be replaced. If the variable is non-critical and not a function parameter, it is replaced with the corresponding variable from the E-stack. If the variable is critical or a function parameter, it is replaced with the corresponding variable from the T-stack. And finally, in case of a function-static variable, the variable is replaced with the corresponding static variable with file scope.

3.5.6. Printer

The Printer is the fifth and last stage of the compiler pipeline. It translates abstract syntax into concrete syntax. Like the parser, the implementation of the printer makes use of the Language.C library [57].

Function 3.17: `print ::`
```
    CTranslationUnit  --  an E-code application in ASR
 →  E-Code            --  the E-code application in CSR
```

Code formatting is subject to long disputes in the software engineering community. Due to the E-code debugger, the formatting of the E-code application is not important, as the software developer never has to deal with it directly. Nevertheless, the code is reasonably formatted, as somebody who wants to improve the T-code compiler has still to be able to read it.

3.6. Summary

In this chapter we presented the primary goal of our thesis: a compiler-assisted thread abstraction. Starting from the big picture we explained the main principles as well as the limitations of our approach. This was followed by a conceptual explanation of the translation from threads to events, illustrated by a small but complete example application.

Subsequently we defined what equivalent T-code and E-code applications are, based on the observable behavior of their operational semantics. This enabled us to informally show the correctness of the transformation.

The platform abstraction layer, which we addressed next, connects the generated E-code application with the existing operating system. We specified the API assumed by E-code applications, presented a complete implementation for Contiki as well as a proof-of-concept for TinyOS, and explained how application-specific abstraction layers can be generated.

Finally, we took a detailed, but still abstract view at the compiler pipeline and its five stages. We carefully specified the respective input and output domains while documenting which C99 language features are either supported or reliably rejected.

Overall, the contribution of this chapter was to introduce a comprehensive, yet efficient thread abstraction which is suitable for resource-constrained system such as WSN motes. The key of our approach is to employ a dedicated compiler which translates C99-compliant T-code into equivalent E-code.

4. Debugger: From E-code to T-code

This chapter addresses the second goal of our work, which is enabling fault diagnostics of T-code. To this end, we have both developed the concepts and implemented a prototype of a T-code debugger. This chapter covers the conceptual aspects while Section 5.2 covers the prototype.

Our goal is not to advance the state-of-the-art in source-level debugging. Instead, we want to focus on providing major source-level debugging capabilities for our compiler-assisted thread abstraction (cf. Section 2.1.6). Most existing WSN programming abstractions are not capable of sustaining the abstraction during fault diagnostics (cf. Section 2.2). This forces the user to understand the technical details of the underlying run-time system nevertheless. Worse, the user is additionally faced with the complexity of the implementation of the abstraction, because he or she has to identify faults in the abstract application from observing the run-time behavior of the generated application. We argue that a comprehensive abstraction should sustain the abstraction level for the complete development cycle.

Support for fault diagnostics conceptually requires the compiler to record each translation step. This information enables the debugger to map run-time information back to source code. Figure 4.1 shows the role of the T-code debugger in the overall project structure. The left side is a condensed version of Figure 3.2, the structure of a T-code project. What is new is that both the T-code and the E-code compiler generate additional debugging information along with the actual output. This information is used by the respective debugger on the right side to perform the back-mapping from the low-level execution of the application to its high-level representation. E-code compilers and debuggers are actually already employing this technique successfully, only that they tend to integrate the debugging information into the generated object file [35, 43, 90]. Conceptually there is, however, no difference to store that information in a separate file.

Just like T-code and E-code compilers are chained, so are E-code and T-code debuggers, i.e., what is the higher abstraction level for the E-code debugger is the lower abstraction level for the T-code debugger. Overall, the T-code debugger executes and controls the E-code debugger, which in turn executes and controls the target binary.

Figure 4.2 provides a detailed view of the modular architecture of the T-code debugger. The central component is the Core module, which implements the actual logic of mapping E-code execution to T-code source. It is interacting with three other components: a front-end, the T-code compiler, and a back-end.

The Front-end interface, as described in Section 4.1, is a set of commands that the Core module provides to a front-end that is controlled by a user. A possible client is, for instance, a *graphical user interface* (GUI) like the one we implemented for the prototype (cf. Section 5.2.3.1). An obvious alternative would be a *command line interface* (CLI) similar to what GDB provides. User interaction can also be simulated by a test suite front-end, which is what we make use of in the evaluation (cf. Section 6.6).

The debug information file interfaces the execution of the T-code compiler with the Core module. Section 4.2 explains, what kind of information is included in that file and

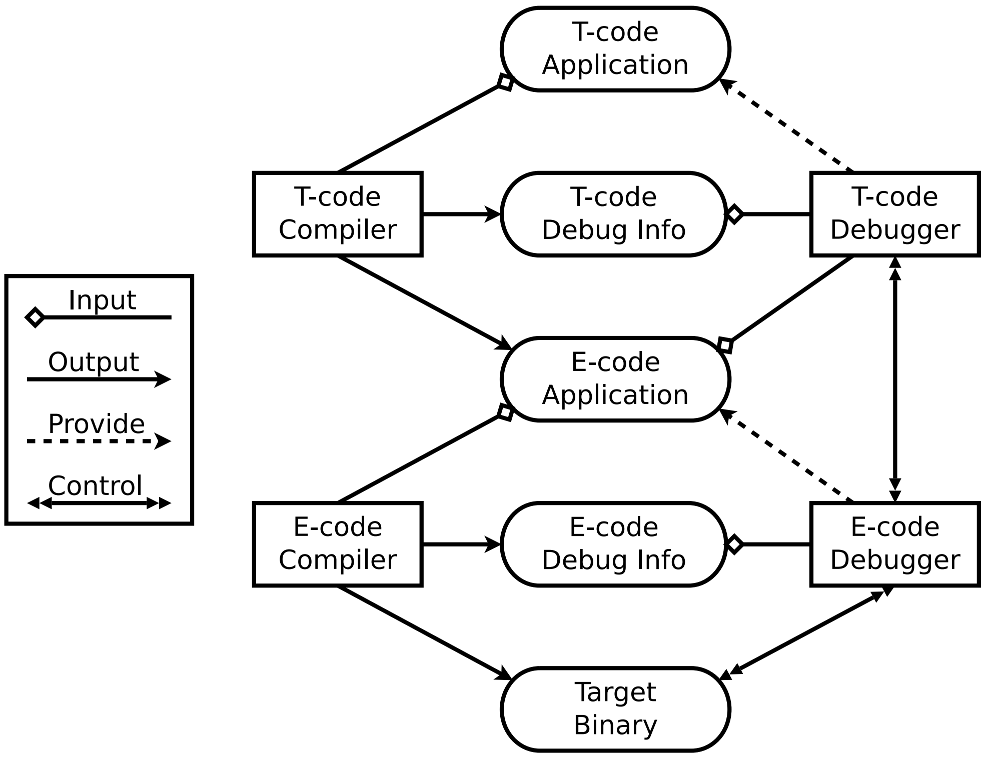


Figure 4.1.: *System overview: debugging:* The compiler generates debugging information which the debugger uses to provide the abstraction level of the input application.

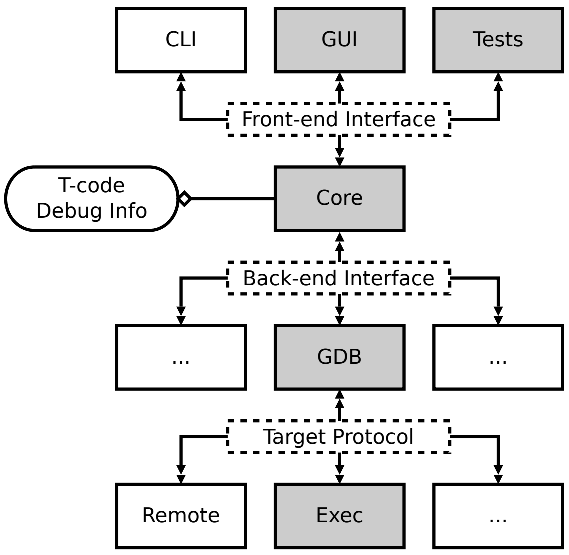


Figure 4.2.: *Debugger architecture:* The Core module provides a User interface and uses debug information generated by the T-code compiler as well as a Back-end interface to interact with the E-code debugger.

how this information is determined within the compiler pipeline. Finally, the Back-end interface provides means to execute and control an E-code debugger. As Section 4.3 will explain, this interface is heavily influenced by the GDB and the only implementation that we provide is a GDB module (Section 5.2.3.3). In principle, however, it should be possible to implement the Back-end interface for other debuggers as well, because it only uses very basic features.

The GDB module executes a GDB instance and controls it via GDB's *Machine Interface* (MI) (Section 27 of [129]) (not shown in Figure 4.2). The GDB on its part supports many architectures and so-called *targets*, i.e., means of connecting to the target binary. For example, the Remote target allows to use a small GDB server that is executed on a device and connected to the GDB instance on the host via a serial cable. The Exec target, in contrast, directly executes a binary file on the host machine. Our evaluation, for instance, makes use of that target (cf. Section 6.6). Of course, the Back-end interface is agnostic to these details.

With the previous sections at hand, Section 4.4 finally explains, how the Core module makes use of all these components to perform its task. Along with Section 4.2 this constitutes the actual innovation of our work. Nevertheless, the other sections provide substantial groundwork.

To focus on our goal, the design and the techniques of the T-code debugger resemble known debuggers like the GDB [129] and Eclipse [132]. Furthermore, the provided features are all limited to the T-code application in general and critical functions in particular. In fact, extending these features to also cover auxiliary functions, utility modules, and libraries only involves copying what E-code debuggers already do.

4.1. Interfacing the User

The Front-end interface defines a set of commands that can be grouped as follows: The first group is covered by Section 4.1.1 and concerns the execution of the debugger and the *inferior* (Section 4.9 of [129]), i.e., the executed target binary. Section 4.1.2 explains how preprocessor directives in the T-code application are dealt with. The next group concerns managing breakpoints, and it is covered by Section 4.1.3. And finally, 4.1.4 investigates evaluating arbitrary expressions containing program variables. All of these commands operate on the abstraction level of the T-code application, i.e., their semantics assume no knowledge about the existence of an E-code debugger.

The set of supported features is minimalistic, but it is sufficient to emulate more advanced features. For example, stepping from one source code row to the next can be achieved by setting a breakpoint at each possible next source code row and continuing the execution. Likewise, watching variables can be replaced by issuing a corresponding query repeatedly. Although such workarounds are hardly acceptable from a usability point of view, they demonstrate that the supported features are conceptually complete.

4.1.1. Execution

Figure 4.3 shows the state machine of the T-code debugger. The initial state is called `waiting`. By issuing the `run` command, the inferior is started and the state changes to `running`. The execution of the inferior can stop for two reasons. The most obvious one is that a breakpoint has been reached, in which case the debugger goes into the `stopped` state. This is, however, not sufficient from a user's point of view. If the execution gets

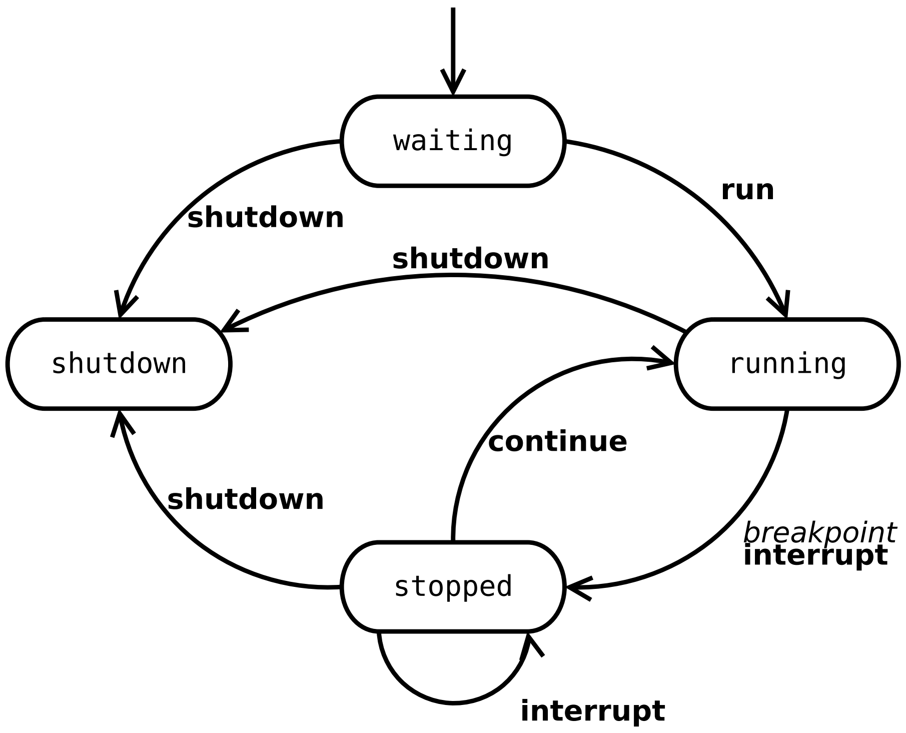


Figure 4.3.: *Debugger state machine:* This figure shows all `states`, **commands**, and *events*.

unanticipatedly trapped in an endless loop, or by chance always takes a control flow path that never hits a breakpoint, there must be a way to regain control. The Front-end interface therefore provides the `interrupt` command which can be issued at any point in time. Just like a breakpoint, it suspends the execution of the inferior and brings the debugger into `stopped` state. Resuming the execution is possible via the `continue` command, which changes the state from `stopped` to `running`. Finally, at any point in time, the `shutdown` command can be issued, which brings the debugger into `shutdown` state.

Figure 4.4 shows how the state of a thread is changed by the debugger's execution commands. At first, a thread is in the `waiting` state. As soon as the PAL calls its thread start function for the first time, it transitions to the `running` state. If it calls a yield point function, it goes into `blocked` state where it stays until the PAL resumes its execution. This behavior is defined by the execution semantics of a T-code application (cf. Section 3.3.1).

There is one additional state during fault diagnostics, which is the state `stopped`. If the debugger stops, either due to a breakpoint or due to the `interrupt` command, a running thread stops as well. Note that according to the T-code semantics there is at most one running thread at any point in time when the debugger is running. For the debugger this implies that there is at most one stopped thread when the debugger is stopped. We refer to the single thread that is either running or stopped as the *current thread*. If the current thread is stopped, then the *current row* is the row number of the instruction that is going to be executed next when continuing that thread. If there is no current thread, then there is no current row either.

Whenever the state of a thread changes, the implementation of the Front-end interface sends an informative update event to the front-end. By displaying this information to the user, he or she is able to follow the execution of the individual threads.

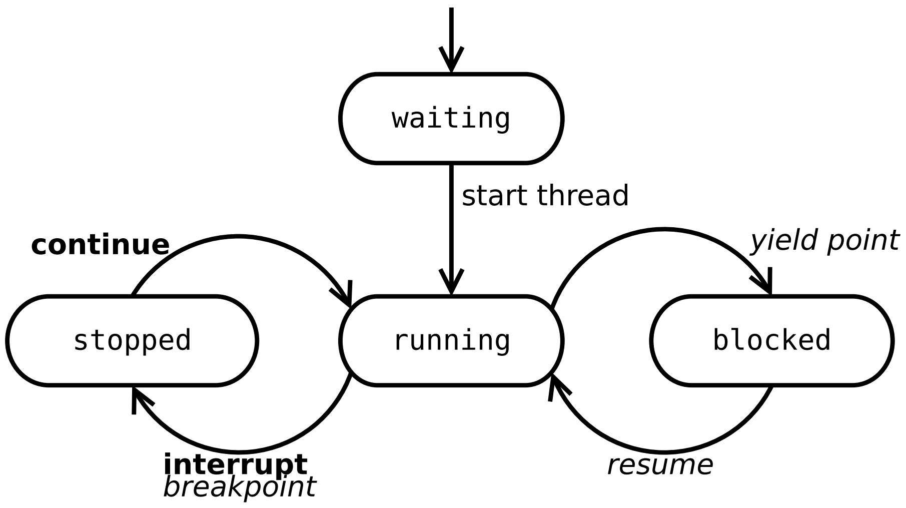


Figure 4.4.: *Thread state machine:* This figure shows all `states`, **commands**, and *events*.

4.1.2. Preprocessing

A T-code application is a preprocessing translation unit that contains preprocessing directives, and the first stage of the compiler pipeline processes these directives to obtain the so-called P-code (cf. Section 3.5.2). As expanding macros can produce significant changes, such as additional critical calls that are not obvious in the T-code, the user might be interested in seeing the P-code while tracing the execution of the application. On the other hand, preprocessing also introduces a lot of external definitions which originate from inclusion directives. Additionally, some macros expand to rather large and confusing code fragments. This suggests to not show P-code to the user.

To support both use cases, the Front-end interface provides a function that translates source code rows from T-code to P-code and vice versa. Actually, the Front-end interface operates on P-code rather than on T-code, as the factual transformation from threads to events is a translation from P-code to E-code. So, depending on the user's interactions, the front-end has to perform a translation.

For instance, the GUI front-end, which is part of the prototype implementation (Section 5.2.3.1), makes use of this functionality by automatically scrolling the view ports for T-code and P-code to always show corresponding source code rows. This allows the user the seamlessly switch between T-code and P-code as needed.

4.1.3. Breakpoints

A breakpoint is an annotation of a source code row which stops the execution of the inferior if the control flow reaches that row. In a T-code application, each thread has its own control flow, which is why breakpoints are augmented with a *thread filter*. A thread filter is a set of thread IDs, and a *thread ID* is a unique number that identifies a thread. The ID of a thread is equal to the source code order of the corresponding thread start function.

A breakpoint causes a stop only if the ID of the current thread is included in the thread filter of that breakpoint. This is useful for re-entrant functions, because it allows to ignore context switches and focus on a specific thread instead. If a breakpoint is *unreachable* for one of the threads of its thread filter, it cannot be added. This feature is supposed to reveal apparent misconceptions early. It is, however, only supported for breakpoints in critical functions.

Besides the thread filters that are specific to breakpoints, there is also a *global thread filter*. A breakpoint causes a stop only if the ID of the current thread is actually included both in the breakpoint's thread filter and in the global one. This allows to quickly restrict the debugger's view to a specific thread without changing all breakpoints. See Section 6.6 for a usage example.

Breakpoints can, of course, not only be added, but also listed and removed. They are identified by a unique number that is assigned to them when added. Also, issuing any of these commands is only possible if the debugger is waiting or stopped. None of this is really interesting in itself, so we avoid the details here.

4.1.4. Variables

In order to query the state of the application, the user can create arbitrary expressions that may involve application variables. The Front-end interface provides a command to evaluate such expressions in the context of the current execution and return the result value. This evaluation is sensitive to both the current thread and the surrounding scope of the current row for two reasons. First, inner scopes can shadow identifiers declared in surrounding scopes. And second, each local variable of a re-entrant function is instantiated once for each invoking thread. Evaluating expressions is only possible when the debugger is stopped, because the current row is not defined otherwise.

4.2. Interfacing the T-code Compiler

As one might anticipate from the previous section, the implementation of the Front-end interface necessitates both static and run-time information of the application. As Figure 4.2 already showed, run-time information is provided by the back-end, while the static information has been determined and stored into the debug information file by the T-code compiler. This section investigates the details of the latter.

While advanced debugging formats such as DWARF [35] support a wide range of different languages and compilers, the debugging information presented in the following is very specific to the compiler described in the previous chapter. This particularly means that both the debugging information and the debugger might have to be adapted if a future compiler employs more aggressive optimizations, for instance.

The documentation of the compiler pipeline in Section 3.5 is actually simplified, as it does not cover the debugging aspect. Figure 4.5 completes Figure 3.12 by actually showing the Debug stage. It is responsible for compiling the debugging information that has been collected by previous stages of the compiler pipeline into the format that is expected by the T-code debugger. How this information is actually used by the Core module is not discussed until Section 4.4. Also, the storage format and other details are not specified until Section 5.2.2.

The following sections introduce the pseudo functions and pseudo data types of the Debug stage. Section 4.2.1 and Section 4.2.2 explain how T-code, P-code, and E-code rows can be converted into each other. Translating arbitrary T-code expressions to E-code expressions involves replacing variables and is investigated by Section 4.2.3. Finally, Section 4.2.4 completes the specification of the debug information with additional auxiliary information. These sections also provide updated specifications of both the pseudo functions of the compiler pipeline and the various code representations.

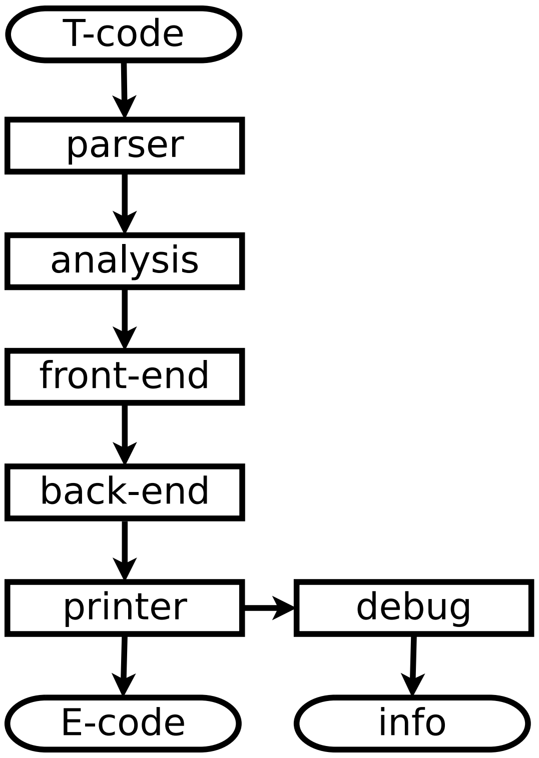


Figure 4.5.: *Compiler pipeline with debugging support:* The actual compiler pipeline has an additional final Debug stage.

4.2.1. Mapping T-code and P-code Rows

As discussed in Section 4.1.2, the Front-end interface provides functions to translate T-code rows to P-code rows and vice versa. To actually implement these functions, the Core module needs the following information.

Data 4.1: `MapTP` =

```
T-Row              --  the maximum row number of the T-code
P-Row              --  the maximum row number of the P-code
[(T-Row, P-Row)]   --  a list of matching T-code and P-code rows
```

The semantics of the first two values is obvious and determining them is trivial. Concerning the third one, the semantics are as follows: The list contains tuples of T-code and P-code rows and is sorted in ascending order of the former. Each tuple defines a section in both the T-code and the P-code, which means that all subsequent rows match consecutively until the beginning of the next section. While for a given T-code row there is always a matching P-code row, the same does not necessarily hold in the other direction, as P-code usually contains a lot of lines that have been included from headers.

Before a simple example can clarify this, we need to understand how preprocessing works. C99 specifies seven different kinds of preprocessor directives [C99: 6.10]. The most relevant ones are:

1. *conditional inclusion*: depending on the value of a conditional constant expression portions of the translation unit may be skipped.

2. *source file inclusion*: this directive is replaced by the complete contents of a source file or *header* [C99: 7.1.2-1].

```
 1  #include <stdio.h>
 2
 3  int function() {
 4    return 0;
 5  }
 6
 7  #include <stddef.h>
 8
 9  void print() {
10    printf("%d\n", function());
11  }
```

(a) T-code

T-code row	P-code row
1	4
2	847
8	858

(b) mapping

```
  1  # 1 "input.c"
  2  # 1 "<built-in>"
  3  # 1 "<command-line>"
  4  # 1 "input.c"
  5  # 1 "/usr/include/stdio.h" 1 3 4
  6  # 28 "/usr/include/stdio.h" 3 4
     ≈                                                                  ≈
845  # 936 "/usr/include/stdio.h" 3 4
846
847  # 2 "input.c" 2
848
849  int function() {
850    return 0;
851  }
852
853  # 1 "/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stddef.h" 1 3 4
854  # 149 "/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stddef.h" 3 4
855  typedef long int ptrdiff_t;
856  # 323 "/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stddef.h" 3 4
857  typedef int wchar_t;
858  # 8 "input.c" 2
859
860  void print() {
861    printf("%d\n", function());
862  }
```

(c) P-code

Figure 4.6.: *Mapping between T-code rows and P-code rows:* CPP line markers tie input and output rows. The input T-code file is called "input.c" in this example.

3. *macro replacement*: defining *macro names* and *replacement lists* that cause "each subsequent instance of the macro name to be replaced by the replacement list" [C99: 6.10.3-9].

Preprocessing resolves these directives, resulting in a translation unit that does not contain any directives or macro names any more. A lot of details of this process are, however, not specified by C99. For example, it is both left open what "skipping" means and what should happen to source code lines that contain a directive other than a source file inclusion directive. We therefore have to resort to an implementation-dependent solution to map T-code and P-code rows as follows:

The *GCC preprocessor* (CPP) [128] emits special *line markers* (Section 9 of [128]) using the *non-directive syntax* [C99: 6.10.1-3]. Such a line marker particularly contains the file name and the line number from which the subsequent lines originate. If the referred file is the translation unit of the T-code application, then the indicated line number is a T-code row, and the line number of the line marker itself is the corresponding P-code row. The only exception is the very first line marker which simply specifies the name of the T-code input file.

Figure 4.6 shows an example which shows how entries in the mapping table correspond directly to line markers. In this example, T-code row 4, for instance, belongs to the second section and maps to P-code row 849, as $847 + (4 - 2) = 849$. In contrast, there is no mapping for P-code row 7, for instance, because $1 + (7 - 4) > 2$, which is the start of the next section.

The Debug stage provides the following function which determines the mapping between T-code rows and P-code rows.

Function 4.1: `map_t2p ::`
```
    T-Code  --  the T-code application in concrete syntax
  → P-Code  --  the preprocessed T-code application in concrete
                syntax
  → MapTP   --  T-code row ↔ P-code row
```

Both T-code and P-code are obtained from the output of the `preprocess` function, as this functions actually is defined as follows:[1]

Function 4.2: `preprocess′ ::`
```
    Path                --  T-code application file
  → String              --  preprocessor command line
  → Either              --  either
      [Error]               --  a list of errors, or
      (T-Code, P-Code)      --  the contents of the file specified
                                by Path, and the output from the
                                preprocessor
```

As already mentioned, `map_t2p` assumes that the CPP has been used to turn T-code into P-code. The reason why CPP keeps track of source line origins is that this information is needed to generate debugging information for the GDB. As any reasonable C tool chain entails a debugger, any preprocessor has to record the same mapping in one way or another. We are therefore confident that adapting `map_t2p` to other preprocessors is always possible.

4.2.2. Mapping P-code and E-code Rows

Being able to translate P-code rows to E-code rows and vice versa is a major capability of the Core module. As Section 4.4 will elaborate on, this is, for instance, needed to turn T-code breakpoints into E-code breakpoints and to determine the current T-code row from an E-code *backtrace* (Section 8.2 of [129]). Both cases consider only rows that could potentially become the current row. Simply stated, this concerns rows that contain either a statement other than a compound statement or a null statement, or a declaration with an initializer.

In contrast to the previous section, P-code rows and corresponding E-code rows do not constitute a one-to-one mapping. Instead, a single P-code row can yield multiple E-code rows for the following two reasons. First, the translation from threads to events turns some input statements such as critical calls into a sequence of E-code statements (cf. Section 3.5.4.5), and each statement tends to be put on a separate row by the printer. And

[1]The prime symbol indicates an update to the function.

second, re-entrant critical functions are inlined into multiple thread execution function (cf. Section 3.5.5.2). To cover all of these possibilities, the T-code compiler generates for each P-code row a mapping as follows:

If the P-code row corresponds to an auxiliary function, then it maps to a single E-code row. This is correct because auxiliary functions are not altered by the translation. If the P-code row corresponds to a critical function instead, then it maps to a different list of E-code rows for each thread calling that function. Also, rows containing a call to a yield point function are flagged, as this information is needed by the Core module to trace the execution of the T-code threads (Section 4.4.2). This leads to the following data structure:

Data 4.2: `MapPE` =
```
[(              --  a list of tuples containing
   Location,        --  a code location, and
   [E-Row]          --  a list of E-code rows
)]
```

Data 4.3: `Location` =
```
P-Row            --  a P-code row number
Maybe ThreadID   --  a threadID or nothing
Boolean          --  call to a yield point function?
```

The information in `MapPE` serves four purposes. First, for auxiliary functions, it can be used to query the matching E-code row for a given P-code row. In this case, there is no thread ID, the list of E-code rows contains only one element, and the Boolean flag is always `false`. Second, for critical functions, it can be used to retrieve the list of matching E-code rows for a given tuple of P-code row and thread ID. Third, it can be used to obtain the E-code row for each call to a yield point function. And last, for a given E-code row, it can be used to find the matching P-code row, if existent, and, if applicable, the corresponding thread.

In order to determine this mapping, each translating step of the compiler pipeline has to record which output statement originates from which P-code row. Also, the printer has to log which statement was rendered to which row of the E-code. Merging this information yields the required mapping from P-code rows to T-code rows.

The `Location` data type bridges the gap between P-code and E-code. While the contained row number is a P-code row number, the Boolean flag marks calls to yield point functions which only exist in the E-code. This means, if the flag is set, then the corresponding list of E-code rows holds a single E-code row only, and that row contains a call to a yield point function.

Collecting the debug information necessitates to extend the code representations used within the pipeline. Section 4.2.2.1 therefore provides an update to the specifications from Chapter 3. Likewise, the signature and the functionality of some pipeline functions change as well, which is covered by Section 4.2.2.2. Also, the Debug stage adds a new function which generates the required mapping. Section 4.2.2.3 completes the discussion with a small, illustrative example.

4.2.2.1. Representations

The compiler pipeline makes use of two distinct representations, the abstract syntax (cf. Appendix A) and the intermediate representation (cf. Section 3.5.1). In order to collect debugging information while processing, the compiler pipeline makes use of the fact that most AST nodes can be attributed with user-defined values.

The `parse` function returns an AST which is attributed with *node infos* (NI).

Data 4.4: `NodeInfo` =
```
Path                 --  the file name of the original source
                         file or header
(T-Row, T-Column)    --  start of the annotated node in the
                         source file or header
(T-Row, T-Column)    --  end of the annotated node in the source
                         file or header
```

The parser, which is a third-party implementation, actually fails to adequately determine column numbers in the presence of macro names, which is why we do not use this information. Also, the reported row numbers are T-code row numbers, which is what a suspected user of that library would probably expect. For our purpose, we are, however, interested in the P-code row numbers instead, as the Front-end interface operates on them. Nevertheless, the compiler pipeline consistently uses T-code row number until the Debug stage converts them into P-code row numbers via `MapTP`.

Early in the compiler pipeline, node infos are turned into *enriched node infos* (ENI), which come in three flavours. First, for AST nodes that belong to auxiliary functions, the enriched node info is just a wrapper around a node info. This is a trivial case and is thus not discussed any further. Second, some AST nodes are generated without having an origin in the T-code and are thus attributed by a special *undefined node info* (UNI). Examples of such nodes include T-stacks, declarators of thread execution functions, the computed `goto` statement from the prologue of these functions, etc. Finally, for AST nodes that belong to critical functions, the enriched node is defined as follows:

Data 4.5: `EnrichedNodeInfo` =
```
T-Row      --  start row of the annotated node
ThreadId   --  the ID of the thread whose execution function
               contains the annotated node
Boolean    --  call to a yield point function?
```

The latter two values are set by the `threads` function, while the first one is kept up-to-date by each involved stage of the compiler pipeline.

The intermediate representation is also able to attribute its nodes. When generating the IR, the `basic_blocks` function copies the ENI from the corresponding AST node to the IR node. Likewise, the `threads` function attributes each AST node with the ENI of the corresponding IR node. By these means the debugging information is conserved when translating from abstract syntax to IR and vice versa. Stages of the compiler pipeline that operate on the IR have therefore no concern with debugging information.

4.2.2.2. Pseudo Functions

The `desugar` function is the first stage of the compiler pipeline that actually replaces statements and introduces new ones. It therefore is the first function whose signature changes to consider debugging information.

input pattern	attributed output

while (condition) }
  body
}
	label1$_{UNI}$: ;
	if$_{eni(while)}$ (!(condition))
	  goto$_{eni(while)}$ label1;
	body
	goto$_{eni(while)}$ label1;

switch (expression){
  case const1:
    body1
  default:
    body2
}
	int var1 =$_{eni(switch)}$ expression;
	if$_{eni(switch)}$ (var1 == const1)
	  goto$_{eni(switch)}$ label1;
	goto$_{eni(switch)}$ label3;
	label1$_{UNI}$: ;
	body1
	label2$_{UNI}$: ;
	body2
	label3$_{UNI}$: ;

Figure 4.7.: *Tracking row numbers:* eni is a function that turns the node info of its parameter into an enriched node info by copying the row information.

Function 4.3: desugar′ ::
```
    [CBlockItem NI]       --  the body of a critical function
 →  (                     --  a tuple consisting of
       [CStatement ENI],      --  the translated body of the input
                                  function, and
       [Variable]             --  the list of new IR variables
    )
```

In addition to its basic functionality, desugar turns NI attributes into ENI attributes with adequate row information. This becomes explicit by the type variables NI and ENI that complete the polymorphic AST types.

Figure 4.7 augments two examples of substituted statements from Table 3.3 by showing how the new statements are attributed. A basic statement label is attributed by the UNI, because it has a null statement attached. This row is therefore never really executed, so tracking it is not necessary. All the other new statements are attributed with the T-row number of the AST node which they substitute. Likewise, goto statements that substitute continue or break statements are attributed with the row number of the latter.

If short_circuit replaces a statement that contains a short-circuit evaluation with a list of statements that emulate that behavior, the latter are attributed with the ENI of the former.

Function 4.4: short_circuit′ ::
```
    [CStatement ENI]      --  the body of a critical function
 →  (                     --  a tuple consisting of
       [CStatement ENI],      --  the translated body of the input
                                  function, and
       [Variable]             --  a list of new IR variables
    )
```

If `normalize` replaces a statement that contains a critical call with a list of statements that contain critical calls only in normal form, the latter are attributed with the ENI of the former.

Function 4.5: `normalize′ ::`
```
    [CStatement ENI]      --  the body of a critical function
 →  (                     --  a tuple consisting of
      [CStatement ENI],      --  the translated body of the input
                                 function, and
      [Variable]             --  a list of new IR variables
    )
```

As described in Section 3.5.4.5, the `basic_blocks` function introduces, among other things, explicit `return` nodes if there is a code path in the input function that reaches the closing brace before reaching a `return` statement. These explicit `return` nodes will become `return` or `goto` statements in the E-code, and the question is, which P-code row should be assigned to them. The obvious choice, which is also implemented by the GDB, is to use the P-code row that contains the closing brace. Therefore, `basic_blocks` receives an extra argument, which is an ENI pointing to the row of that brace.

Function 4.6: `basic_blocks′ ::`
```
    [CStatement ENI]  --  the body of a critical function
 →  ENI               --  referring to the function's closing
                          brace
 →  Body              --  the translated body of the function
```

The `threads` function both turns IR nodes (T-code) into AST nodes (E-code) and creates additional AST nodes. Those additional nodes are always attributed by the UNI, while nodes that originate from IR nodes are attributed with the ENI of the corresponding IR nodes.

Function 4.7: `threads′ ::`
```
    [Function]         --  the list of critical functions in IR
 →  [CDeclaration NI]  --  the E-stacks
 →  [CFunctionDef ENI] --  the thread execution functions
```

Each `CFunctionDef` is the thread start function of one thread, and `threads` attributes each of its child nodes with the thread ID of the corresponding thread. Also, the AST nodes that represent calls to yield point functions are flagged.

The `print` function turns the abstract syntax of E-code into its concrete syntax. In addition to that, it also collects a *printer log*.

Function 4.8: `print′ ::`
```
    CTranslationUnit ENI  --  an E-code application in ASR
 →  (                     --  a tuple consisting of
      E-Code,                --  the E-code application in CSR,
                                 and
      PrinterLog             --  the printer log
    )
```

Data 4.6: `PrinterLog =`
```
[(        --  a list of tuples containing
   ENI,       --  (T-Row, ThreadID, Boolean), and
   E-Row      --  the E-code row number
)]
```

The attributes of all AST nodes are either the UNI or they are an ENI that refers to the T-code row from which that node originates. In the latter case, the `print` function associates the E-code row to which an AST node was rendered with the ENI of that node. While the printer is a third-party implementation, we had to extend it to actually support logging. Section 5.1.3.2 describes the details.

Finally, the Debug stage provides a function that turns the printer log into a mapping from P-code rows to T-code rows. It uses `MapTP` to translate the T-code rows of the enriched node infos of the printer log to P-code rows.

Function 4.9: `map_p2e ::`
```
   PrinterLog  --  the printer log
→  MapTP       --  T-code row ↔ P-code row
→  MapPE       --  P-code row ↔ E-code row
```

4.2.2.3. Example

Figure 4.8 shows a small example that includes the re-entrant function `c`, the auxiliary function `f`, the blocking function `b` and the two thread start functions `s1` and `s2`. Just like Figure 3.6, this example omits implementation details like the applied scheme of unique names.

For instance, P-code row 6 is mapped to various E-code rows. This is because, first, the function `c` is re-entrant and is thus embedded into both thread execution functions. And second, this is because row 6, just like row 9 and 12, contains a critical call, which leads to a sequence of E-code rows. In each case, the single row that contains the call to the yield point function `b` is flagged. The mapping for the rows 7, 10, 13 is caused by the explicit `return` nodes added by the `basic_blocks'` function.

The same is, however, not done for row 4, as it belongs to an auxiliary function. As already mentioned at the beginning of this chapter, our current compiler pipeline does not extend its functionality towards auxiliary functions, so the necessary analysis to identify implicit `return` statements is not performed for them. Instead, the Back-end simply wraps the NI of each AST node that could become part of the current row into an ENI. This leads to the mapping between the P-code row 3 and the E-code row 25.

4.2.3. Mapping Variables

Figure 4.8 also shows how local variables of the P-code become global variables in the E-code. If, for instance, the current row is 6 and the current thread is the one of `s1`, then the P-code expression `i` refers to the E-code expression `tstack_s1.frames.c.i`. Similar translations are necessary for E-stack variables as well as function-static variables, which is why the T-code generates an according translation map.

```
 1  __attribute__((tc_block)) void b(int i);
 2  int f(int i) {
 3    return i + 1;
 4  }
 5  void c(int i) {
 6    b(f(i));
 7  }
 8  __attribute__((tc_thread)) void s1() {
 9    c(23);
10  }
11  __attribute__((tc_thread)) void s2() {
12    c(42);
13  }
```

(a) P-code

Location			
PRow	**ThreadID**	**Boolean**	**[ERow]**
3	n/a	f	25
6	0	f	37, 38, 40
6	0	t	39
7	0	f	42
9	0	f	31, 32, 33
10	0	f	35
6	1	f	54, 55, 57
6	1	t	56
7	1	f	59
12	1	f	48, 49, 50
13	1	f	52

(b) mapping

```
 1  typedef struct {
 2    void * cont;
 3    int i;
 4  } tframe_b_t;
≈                                                          ≈
24  int f(int i) {
25    return i + 1;
26  }
27  void thread_0(void* cont)
28  {
29      if (cont) goto* cont;
30      s1_1: ;
31      tstack_s1.frames.c.i = 23;
32      tstack_s1.frames.c.cont = &&s1_2;
33      goto c_1;
34      s1_2: ;
35      return;
36      c_1: ;
37      tstack_s1.frames.c.frames.b.i = f(tstack_s1.frames.c.i);
38      tstack_s1.frames.c.frames.b.cont = &&c_2;
39      b(&tstack_s1.frames.c.frames.b);
40      return;
41      c_2: ;
42      goto * (tstack_s1.frames.c.cont);
43  }
44  void thread_1(void* cont)
45  {
46      if (cont) goto* cont;
47      s2_1: ;
48      tstack_s2.frames.c.i = 23;
49      tstack_s2.frames.c.cont = &&s2_2;
50      goto c_1;
51      s2_2: ;
52      return;
53      c_1: ;
54      tstack_s2.frames.c.frames.b.i = f(tstack_s2.frames.c.i);
55      tstack_s2.frames.c.frames.b.cont = &&c_2;
56      b(&tstack_s2.frames.c.frames.b);
57      return;
58      c_2: ;
59      goto * (tstack_s2.frames.c.cont);
60  }
```

(c) E-code

Figure 4.8.: *Mapping P-code rows to E-code rows:* A small example with re-entrant and auxiliary functions.

Data 4.7: VarMap =
```
[(                    --  a tuple list consist. of
   (P-Row, P-Row),     --  the first and the last row of a
                           scope, and
   [Rename]            --  a list of re-namings of variables
)]
```

Data 4.8: Rename =
```
(                   --  a tuple consisting of
   String,          --  the P-code name, and
   FQN              --  the E-code “name”
   Maybe ThreadID   --  possibly a corresp. thread
)
```

The first stage of the compiler pipeline that deals with variables is the `collect` function. It not only removes declarations from the function body, but it also renames them to guarantee their uniqueness. To enable scope-based queries in the T-code debugger, the function `collect` additionally augments a newly created variable with information about the scope of its visibility.

Data 4.9: Variable′ =
```
String          --  original name
CDecl           --  renamed declaration
[Flag]          --  flags
(T-Row, T-Row)  --  scope (first row, last row)
```

The scope information is taken from the NI of the respective current surrounding `CCompound` statement, which is why it is represented as T-code rows rather then P-code rows. Just as in Section 4.2.2, the Debug stage will determine the corresponding P-code rows later.

While subsequent stages of the compiler pipeline introduce additional variables, none of them actually exists in the original T-code, which means they will not be queried by the user of the T-code debugger. So, the next involved stage is the Back-end which introduces the E-code variables. For each variable that has been gathered by `collect`, it records its so-called *fully qualified name* (FQN) in the E-code.

The FQN of a variable is a `CExpression` in concrete syntax that addresses that variable uniquely. For example, the FQN of a critical variable could be `tstack_s1.frames.c.i`, while the FQN of a non-critical variable could be `estack.wait.now` (cf. Figure 3.6). These two examples show that the same variable can have multiple FQNs, one for each thread that invokes the function that declares that variable. Finally, the FQN of a function-static variable simply is its unique name, as the Back-end generates identifiers with file scope for them. In this case, there is no associated thread, as all threads see the same variable instance (cf. Section 3.3.1).

More precisely, a minor function of the Back-end stage takes care of function-static variables, while local variables are handled by `threads`.

Function 4.10: `threads″ ::`
```
    [Function]              --  critical functions
 →  [CDeclaration NI]       --  E-stacks
 →  (                       --  a tuple consisting of
       [CFunctionDef ENI],      --  thread exececution functions,
                                    and
       [(Variable, FQN)]        --  variables and their
                                    corresponding FQNs
    )
```

With this information at hand, the Debug stage can finally create the mapping for variables.

Function 4.11: `map_var ::`
```
    MapTP              --  the mapping between T-code and P-code
                           rows
 →  [(Variable, FQN)]  --  the list of variables and their
                           corresponding FQN
 →  VarMap             --  T-code variable ↔ E-code FQNs
```

4.2.4. Debugging Information

The debug information shared between T-code compiler and T-code debugger actually contains additional auxiliary information. This entails references to both the T-code and the E-code file, the P-code itself and information about the T-code threads.

The T-code file is used by the debugger front-end to retrieve the T-code in order to display it to the user. Likewise, the P-code, as returned by the `preprocess` function, is intended to be displayed by the front-end. The E-code, in contrast, should not be shown to the user, because the purpose of the T-code debugger is exactly to hide the details of the E-code abstraction level. Nevertheless, when developing the debugger itself, seeing the E-code is very useful, which is why we left a reference to it in the debug information. The data types `MapTP`, `MapPE`, and `VarMap` have been covered by the previous sections. What is left is the information about the thread functions. This information is needed to trace the execution of the threads, as Section 4.4.2 will explain. From this information the T-code debugger can additionally deduce the number of threads and their IDs.

Data 4.10: `DebugInfo =`
```
   Path        --  the T-code file
   String      --  the P-code
   Path        --  the E-code file
   MapTP       --  T-code row ↔ P-code row
   MapPE       --  P-code row ↔ E-code row
   VarMap      --  T-code variable ↔ E-code FQNs
   [(          --  a list of tuples consisting of
      String,      --  name of a thread start function
      String       --  name of the corresponding thread execution
                       function
   )]
```

4.3. Interfacing the E-code Debugger

The Back-end interface serves the communication between the Core module and the respective back-end. As already mentioned, this interface is heavily influenced by the GDB back-end. Nevertheless, the Back-end interface is conceptually agnostic to the respective implementation, as it only uses basic features that any other back-end should be able to provide as well.

As described in Section 4.1, the user has to be able to interrupt the execution of the inferior at any point in time. Since this entails asynchronism, the Back-end interface consists of a set of commands and events. Similar to the Front-end interface, the commands and events can be grouped into execution, breakpoints, and evaluation of expressions.

Executing the back-end and the inferior involves the commands `setup`, `run`, `interrupt`, `continue`, and `shutdown`. While the `setup` command is issued by the Core module on start-up, all the other commands are directly related to the equally named Front-end commands. So, the overall state machine of the back-end resembles the state machine of the Core module as depicted in Figure 4.3. All of these functions are synchronous.

Also similar to the Front-end interface, breakpoints are identified by unique numbers. When a new breakpoint is set, the `set_breakpoint` function expects a *location* and returns the breakpoint number. There are various possibilities to specify a location, among which only file name plus function name and file name plus row number are currently used. The function `remove_breakpoints` expects a possibly empty list of breakpoint numbers which are then removed. The Back-end interface provides no function to query the list of breakpoints. Instead, the Core module is expected to keep track of the added and removed breakpoints by itself. Whenever the control flow of the inferior hits a breakpoint, a `BreakpointHit` event containing the number of the respective breakpoint is sent to the Core module. As the corresponding state transition from `running` to `stopped` is asynchronous, it may interleave with invocations of the `interrupt` command.

The function `evaluate` takes a C expression in concrete syntax and returns either an error or the value of that expression as a string. As already mentioned in Section 4.1.4, evaluating expressions is sensitive to the current state of the execution, particularly the current row. In order to make this information accessible to the Core module, the Back-end interface provides the `backtrace` function which returns the current call stack.

Data 4.11: `Stack` =
```
[Frame]  --  the list of stack frames
```

Data 4.12: `Frame` =
```
String  --  the name of the respective function
Path    --  the name of the respective source file
E-Row   --  the source code row
```

The head of the list of frames is the current stack frame. The source code row of that frame is the current row, while the rows of the other frames are the respective row of the function call which caused the previous frame of the list.

4.4. The Core Module

The major duty of the Core module is to map E-code run-time information back to the T-code abstraction. Technically stated, it uses the debug information from Section 4.2 and the Back-end interface from Section 4.3 to implement the Front-end interface as described in Section 4.1. Its central data structure is the following.

Data 4.13: `Core =`
```
State         --  waiting, running, stopped, or shutdown
[Breakpoint]  --  the list of back-end breakpoints
[(Int, Int)]  --  mapping from front-end to back-end breakpoint
                  numbers
[Int]         --  the global thread filter
[Thread]      --  the list of T-code threads
```

The transitions of the state have been described in Figure 4.3 and the corresponding front-end commands implement just that state machine. The only corner case is the `interrupt` command as it might interleave with a `BreakpointHit` event from the back-end, but handling this case is trivial. The commands and the three data members that involve breakpoints are described in Section 4.4.1. Section 4.4.2 explains how the Core module tracks the execution of the T-code threads by observing the execution of the corresponding thread execution functions. Next, Section 4.4.3 investigates how arbitrary T-code expressions can be evaluated.

Most parts of the Core module involve querying the `MapTP`, the `MapPE`, and the `VarMap` data structures. Although the involved algorithms are not always straight forward, they are still rather intuitive and not very interesting by themselves. We therefore do not cover them here but refer to the source code distribution of our work for details.

4.4.1. Breakpoints

The Core module distinguishes three types of breakpoints. First, a *front-end breakpoint* is a breakpoint that has been added via the Front-end interface. Second, a *back-end breakpoint* is a breakpoint that is added via the Back-end interface. And third, an *internal breakpoint* is a back-end breakpoint that is not a front-end breakpoint. Section 4.4.2 will make use of these to trace the execution of the T-code threads.

While there is one back-end breakpoint for each front-end breakpoint, both types of breakpoints have an independent set of unique numbers. This is necessary to maintain a linearly increasing, and thus intuitive, numbering of front-end breakpoints despite of possible internal breakpoints. Therefore, the Core module maintains not only a list of back-end breakpoints but also a mapping from front-end breakpoint numbers to back-end breakpoint numbers.

The implementation of the front-end breakpoint commands is straight forward, as it basically requires only to translate the P-code row to an E-code row and add the back-end breakpoint. Implementing the additional analysis for unreachable breakpoints is possible, as the `MapPE` data structure lists the thread IDs of all threads whose control flow might execute a given E-code row. If a `BreakpointHit` event is received, the reaction depends on the involved thread filters. If the current thread is included in the breakpoint's thread filter as well as in the global thread filter, then the state of both that thread and the Core transitions to `stopped`. Otherwise, the `continue` command is issued to the back-end, i.e., this breakpoint is effectively ignored.

4.4.2. Thread Execution

In order to keep track of the execution of the T-code threads, the Core module installs a series of internal breakpoints. First, one for the entry of each thread execution function via locations based on file name plus function name. And second, one for each call of a yield point function via locations based on file name plus row number. The list of yield point functions is part of the debug information, and the list of all calls to yield point functions can be extracted from the `MapPE` field of the debug information.

Every time a breakpoint of the first category is hit, the Core changes the state of the corresponding thread from either `waiting` or `blocked` to `running` (cf. Figure 4.4). Likewise, every time a breakpoint of the second category is hit, the Core changes the state of the corresponding thread from `running` to `blocked`. In any case, the `continue` command is issued to the back-end to resume the execution of the inferior, as these internal breakpoints are not supposed to become visible to the user. The state of a thread is stored in the following data type.

Data 4.14: `Thread` =
```
Int          --  the thread ID
String       --  the thread start function
State        --  waiting, running, blocked, or stopped
Maybe P-Row  --  the current row if existent
```

Each time the status of a thread changes, the front-end is informed to make this change visible to the user, while the name of the thread start functions is supposed to help the user to identify the individual threads. The P-code row is updated whenever the thread changes its state to `blocked` or `stopped`. The current row is thereby obtained by querying the current `Stack` from the back-end and translating the source code row of the top-most `Frame`. As already explained in Section 4.2.2, this fails if the respective E-code row is not included in the mapping. By construction, the only situation where this can happen is if the execution is interrupted. Then, either all threads are blocked, or one thread is indeed running but its current row is still somewhere in the prologue of the thread execution function, i.e., the thread has not really resumed yet.

4.4.3. Evaluating Expressions

Evaluating arbitrary T-code expressions requires a series a consecutive steps. As most groundwork is already established, the Core module overall only correlates the right sources of information with each other.

First of all, as already explained in Section 4.1.4, evaluating expressions depends on the current row. The Core module therefore determines it by querying `backtrace` and translating the source code row of the top-most `Frame` from E-code to P-code. Next, it tries to find a scope in the `VarMap` data structure that encloses the current row. If that fails, then the current row is not within a critical function, which means the expression can be handed over to the back-end unchanged. Otherwise, the variables within the expression have to be translated.

The Core module therefore uses a sub-component of the parser to turn the given expression from concrete syntax to abstract syntax. This syntax tree can now be walked to find and replace all sub-expressions that are in fact variables. Given the current thread, there is at most one matching FQN for each of these variables in the `VarMap` data structure. If there is one, the variable is replaced by a new variable whose "name" becomes that FQN. If there is none, then, by construction, the variable is a global variable and therefore needs no translation. A sub-component of the printer is used to turn the new expression from abstract syntax back to concrete syntax. While, technically, a FQN most of the time is not a valid identifier, the printer is not confused by this and generates a proper expression string nevertheless. This string is finally handed over to the back-end for evaluation.

Evaluating a T-code expression can fail for three different reasons. First, if there is no current row, then the evaluation can obviously not be performed. Second, if the expression is syntactically ill-formed, the parser fails to interpret it. And third, if the expression is semantically wrong, the back-end fails to evaluate the translated expression. Examples of the latter include expressions that involve non-existent variables, or performing array indexing on a variable that has no array type, etc. Any of these cases is reported to the front-end as it is up to the user to resolve the problem.

4.5. Summary

In this chapter we presented the second goal of our thesis: fault diagnostics for our thread abstraction. Starting from the overall setup of T-code and E-code compilers and debuggers, we explained the architecture of the T-code debugger.

We addressed the user interface, covering the basic features that our debugger provides. Subsequently, we explained which information the T-code debugger needs from the T-code compiler, and how the latter had to be extended to actually collect this information. We also specified the interface to the back-end which communicates with the E-code debugger. Finally, we discussed how the previously introduced components are integrated in order to actually perform the back-mapping from E-code run-time to T-code source code.

Overall, the contribution of this chapter was to demonstrate how fault diagnostics can be supported for a compiler-assisted programming abstraction. To focus on this, our debugger heavily imitates known debugger tools and provides only a limited, but conceptually complete feature set.

5. Prototypes

This chapter provides a documentation of core parts of the two prototypes that we have developed in the context of this work. Providing a complete documentation of the implementations is beyond the scope of this thesis. Instead, we focus on the following three aspects and refer to the source code distribution [6] otherwise.

First, we explain the respective overall architecture and provide references to the sections that cover the concepts of the individual modules. Second, we investigate major interfaces that are necessary to extend the functionality of the prototypes and port them to different platforms. And finally, we explain details of a few selected problems with non-trivial solutions that we encountered during the implementation of the prototypes. Section 5.1 investigates Ocram, the T-code compiler prototype, and Section 5.2 covers Ruab, the T-code debugger prototype. Both sections are structured according to these three aspects.

The source distribution involves various programming languages, namely Haskell [4], C [64], Java [45], JavaScript [65], Python [133], and Jinja2 [114]. Some of these languages are required by the given environment of our work, while others reflect our subjective choice of the right tool for the given problem. Ocram and Ruab are implemented in Haskell, because Haskell's expressiveness facilitates quick evaluation of ideas and rapid development of prototypes (cf. Section 2.3). C is obviously used to implement the various case study applications of the evaluation in Chapter 6. Java is the language in which Cooja [150], the network simulator used for the evaluation, is written. Consequently, our Cooja plugin, which collects various measurements during the evaluation, is written in Java as well. The individual experimental runs are scripted by small JavaScript snippets which are executed by Cooja via Rhino[1], a Java-based open-source implementation of JavaScript. Python is used to implement all kinds of infrastructure scripts, such as verification scripts for the evaluation and PAL generator scripts. The latter make in particular use of Jinja2, a template engine for Python. Apart from the short introduction to Haskell in Section 2.3, explaining all these languages goes beyond the scope of this thesis and we refer to the cited documentations instead.

5.1. Ocram: T-code Compiler

This section covers three aspects of the Ocram, our T-code compiler prototype. The first aspect concerns the overall software architecture and is discussed in Section 5.1.1. The second aspect in 5.1.2 addresses key interfaces of Ocram. And finally, Section 5.1.3 highlights a few challenges of the implementation.

module	**section**	**description**
Analysis	3.5.3	perform static analysis
CallGraph	3.5.3	call-graph analysis and query functions
FGL	n/a	extensions to the Functional Graph Library
Filter	3.5.3	enforce global and critical constraints
Types	n/a	data types of the Analsysis module
Backend	3.5.5	generate E-code
EStack	3.5.5.1	generate E-frames and E-stacks
ThreadExecutionFunction	3.5.5.2	generate thread execution functions
TStack	3.5.5.1	generate T-frames and T-stacks
Utils	n/a	common utility functions
Debug	4.2	compile debug information
DebugInfo	4.2	`map_t2p`, `map_p2e`, `map_var`, etc.
Enriched	4.2.2.1	ENI definition and utility functions
Types	n/a	data types of the Debug module
Intermediate	3.5.4	determine the IR
BooleanShortCircuiting	3.5.4.3	perform explicit short-circuit evaluation
BuildBasicBlocks	3.5.4.5	abstract syntax to to IR
CollectDeclarations	3.5.4.1	separate declarations from statements
CriticalVariables	3.5.4.7	distinguish non-critical variables
DesugarControlStructures	3.5.4.2	substitute statements with basic statements
Filter	3.5.4.8	enforce additional constraints
NormalizeCriticalCalls	3.5.4.4	establish normal form of critical calls
Optimize	3.5.4.6	perform basic optimizations
Representation	3.5.1	definition of the IR data types
IO	n/a	perform non-pure I/O operations
Main	n/a	integration
Names	n/a	naming schemes of unique names
Options	n/a	process command line options
Print	5.1.3.2	pretty printing with printer log
Query	n/a	query functions for AST nodes
Ruab	5.2.2	interfacing Ruab
Symbols	n/a	retrieve the identifier of an AST node
Text	n/a	render errors
Util	n/a	utility functions

Table 5.1.: *The Ocram architecture:* The responsibilities of the various modules and sub-modules in alphabetical order.

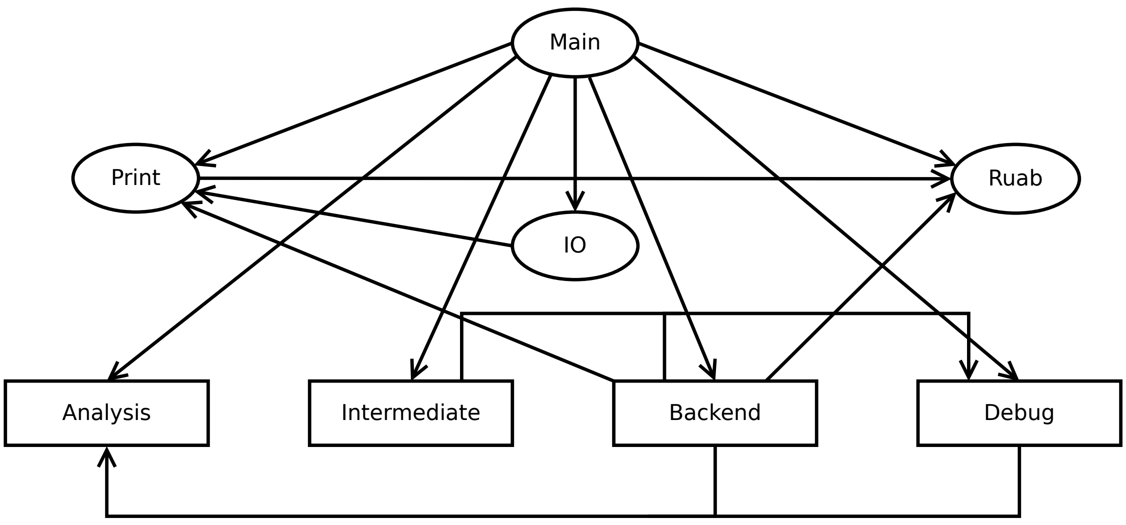


Figure 5.1.: *The Ocram modules:* We only show top-level modules and omit dependencies to infrastructure modules.

5.1.1. Architecture

This section provides an overview of the architecture of Ocram. Table 5.1 provides the complete list of Haskell modules that constitute the T-code compiler, while Figure 5.1 shows an excerpt of the dependency graph of these modules. The concepts behind most modules are described by the referenced sections. Due to the expressiveness of Haskell, the difference between these descriptions and the actual implementation is manageable. Given the background of the these sections, it should therefore be possible to understand the code, given a basic familiarity with Haskell, of course. We therefore refer to the source distribution of this thesis and do not go into further details here. In the following, we will say a few words about the modules whose concepts are not covered in other sections.

The *Functional Graph Library* (FGL)[2] provides data structures and algorithms for inductive graphs, i.e., integer nodes with directed edges, both attributed with user-defined values. The Analysis module makes use of the FGL to represent the call graph and defines the appropriate data types in the Types sub-module. While the FGL includes several helpful graph algorithms like breadth first search and spanning trees, the Analysis module requires a few more. The FGL module therefore particularly implements a loop detection algorithm which is used by the `critical_constraints` function to detect recursive critical functions.

The IO module hosts all operations that involve input or output. That is to say, all the other modules of Ocram contain only pure functions. The non-pure functions of the IO module read input files, invoke external programs, and write output files.

The Names module defines naming schemes used by the various stages of the compiler pipeline. The examples in this thesis do not consider these schemes for clarity. And in fact, due to the T-code debugger which hides E-code details, the only thing that matter is that new identifiers are guaranteed to be unique. As already mentioned in Section 3.5.3, the current implementation uses a reserved prefix to define its own name space of identifiers. Within this name space, each stage defines its own subspace to avoid conflicts with

[1]`https://developer.mozilla.org/en-US/docs/Rhino`
[2]`http://hackage.haskell.org/package/fgl`

```
 1  runCompiler :: IO ()
 2  runCompiler = do
 3    argv                 <- getArgs
 4    prg                  <- getProgName
 5    cwd                  <- getCurrentDirectory
 6
 7    opt                  <- exitOnError "options"   $ options prg cwd argv
 8    (tcode, pcode, tAst)  <- exitOnError "parser"  =<< parse opt
 9    ana                  <- exitOnError "analysis"  $ analysis tAst
10
11    ir                   <- exitOnError "front-end" $ ast_2_ir ana
12    let (eAst, pal, vm)    = tcode_2_ecode ana ir
13    let (ecode, pl)        = render_with_log eAst
14    let di                 = create_debug_info opt tcode pcode ana vm pl ecode
15
16    _                    <- exitOnError "output"  =<< generate_pal opt ana pal
17    _                    <- exitOnError "output"  =<< dump_ecode opt ecode
18    _                    <- exitOnError "output"  =<< dump_debug_info opt di
19    _                    <- exitOnError "output"  =<< dump_pcode opt pcode
20
21    return ()
22
23  exitOnError :: String -> Either [Error] b -> IO b
```

Figure 5.2.: *The Ocram Main module*

others. And finally, each stage numbers new identifiers consecutively, which overall yields distinct unique names.

The Options module defines and processes command line options of the Ocram executable. Possible options concern the paths to input and output files, external tools like the preprocessor and the PAL generator, etc. Invoking the Ocram executable with "–help" prints a documented list of all possible options.

The Query module provides query functions for AST nodes. One function, for instance, returns the parameter list of a function definition. Another function identifies thread start functions via function attributes. Similarly, the Symbols module provides a family of functions that determine the identifier behind a given AST node. This could be a function definition, a variable or an external declaration, for instance. Both of these modules contain functions which could be of general interest, which is why we are planning to contribute them back to the Language.C library.

The Text module defines the text-based output interface of the Ocram compiler. It particularly provides functions to format compiler errors consistently to achieve a user-friendly description of the encountered problem. And finally, the Util module contains various utility functions used throughout the project.

Most modules and sub-modules are paired with an extensive set of unit and integration tests. Theses tests are based on the HUnit[3] and the test-framework[4] libraries. Invoking the Ocram executable with "–test" executes the tests while appending "–help" prints a list of possible command line options to configure their execution.

The Main module integrates all the other modules into the actual Ocram application. Figure 5.2 shows an excerpt, for which Section 2.3.2 actually provides (almost) all required knowledge.

The right hand side of a left arrow is a monadic value; in this case of the I/O monad (lines 3–11, 16–19). The right hand side of an equal sign is, in contrast, a pure value, which is assigned to a pattern of names via **let** (lines 12–14). Functions that can fail have

[3]http://hackage.haskell.org/package/HUnit
[4]http://hackage.haskell.org/package/test-framework

```
schroot -c contiki -p -- \
  msp430-gcc -mmcu=msp430x1611 \
  -DCONTIKI=1 -DUIP_CONF_IPV6_RPL \ # more Contiki-specific macro definitions
  -I. -I$(CONTIKI)/platform/sky -I$(CONTIKI)/cpu/msp430 \ # more include paths
  -DOCRAM_MODE \
  -E -o -
```

Figure 5.3.: *Preprocessing the Contiki applications:* Preprocessing is highly dependent on the environment.

a return type of `Either [Error] b` for any given `b`, sometimes as a pure value (lines 7, 9, 11) and sometimes as a monadic value (lines 8, 16–19). In any case, `exitOnError` takes the result of such functions, fails the compiler in case of a `Left [Error]` and wraps the `Right b` value into the I/O monad otherwise. As the I/O monad executes sequentially, this makes the compiler fail as soon as the first stage of its pipeline fails. The operator =<< is same as >>=, but with arguments interchanged. This is how monadic values are passed to `exitOnError`. Pure values are, in contrast, passed to `exitOnError` via the operator `$`, which is a simple function application with low precedence, thus saving a few braces. The output functions return either a list of errors or unit, which is why the left-hand sides of lines 16 to 19 discard these values via the wild-card pattern `_`.

When we abstract from all these details, the general structure of lines 3 to 19 is the same: the names on the left-hand sides are bound to the values of the right-hand sides. Following the usage of those names reveals the data flow of the Ocram compiler. For example, line 11 invokes the Front-end to receive the intermediate representation of the T-code application. The result value (`ir`) is then passed to the invocation of the Back-end in line 12. The result of this function is the AST of the E-code (`eAst`), the PAL footprint (`pal`), and the `VarMap` (`vm`). These values are in turn used at later stages of the pipeline.

This completes the presentation of the software architecture of Ocram. Please consult our source code distribution [6] for more details.

5.1.2. Interfaces

This section addresses key interfaces of the Ocram architecture that are required to either extend or port the compiler. Section 5.1.2.1 documents the interface to the external preprocessor that turns T-code to P-code. Next, Section 5.1.2.2 specifies the interface to the external PAL generator script. And finally, Section 5.1.2.3 addresses the inclusion of additional optimization passes into the compiler pipeline.

5.1.2.1. Preprocessor

The Parser stage of the compiler pipeline depends on an external C preprocessor to turn T-code into P-code. This section explains the expected interface.

The command line that invokes the preprocessor is passed to Ocram as a command line option. The `preprocess` function of the Parser stage reads the T-code application from the file system and passes it to the standard input of a subprocess as specified by the given command line. The resulting P-code is expected on the standard output of this subprocess. Thus, a minimal command line for the GCC would be “gcc -E -o -”, for instance.

Usually, things are, however, more involved. Figure 5.3 shows an excerpt of the command line of the preprocessor that we use for the evaluation of the Contiki-based case study applications. This example clearly shows why preprocessing is highly depending on

```
#ifdef OCRAM_MODE
#define TC_RUN_THREAD __attribute__((tc_thread))
#define TC_BLOCKING __attribute__((tc_block))
#else
#define TC_RUN_THREAD
#define TC_BLOCKING
#endif

TC_BLOCKING _Bool sleep(int until);
TC_RUN_THREAD void blinky() {
  //...
}
```

Figure 5.4.: *Handling C tools without support for GNU C extensions:* `OCRAM_MODE` is only enabled when generating P-code for Ocram.

the environment. First, the `schroot` command performs a `chroot` system call to change the root directory to the installation of the Contiki distribution. Therefore `msp430-gcc` refers to the MSP430 port of the GCC [79] as installed in that distribution. By specifying the right hardware architecture, the preprocessor automatically includes the correct system headers. In addition, a series of Contiki-dependent macros are specified to perform various configurations. Also, a long list of include directories is required to help the preprocessor in finding the Contiki and the application headers.

While line 6 implements the required interface as described above, line 5 is part of a technique that addresses C tools which do not support GNU C extensions. The idea is to insert the function attributes that mark thread start functions and blocking functions only when Ocram mode is enabled, and to enable that mode only when generating P-code for Ocram. Therefore, all the other C tools are not confronted with function attributes, which are a GNU C extension and might therefore not be supported. Figure 5.4 shows the implementation of this technique.

As this example shows, preprocessing can be quite involved and it is hard to predict the requirements of future platforms Ocram might be ported to. We have therefore chosen the described generic input-output interface via external program to stay flexible.

5.1.2.2. PAL Generator Script

As Section 3.4.4 describes, Ocram depends on the PAL generator script, an external tool that generates an application-dependent platform abstraction layer. This section specifies the interface assumed by Ocram.

The PAL generator script is a program that reads the PAL footprint (cf. Section 3.5.5) from standard input, takes the application footprint (cf. Section 3.4.4) as command line arguments, and prints the PAL to standard output. More precisely, the PAL footprint is a list of external declarations in concrete syntax that are supposed to be included in the PAL right after all include directives. These external declarations define the E-frames, the yield point functions, and the T-stack and T-frames of all critical functions that are used by the application. The application footprint is given as one list of comma-separated names of invoked yield point functions per thread. Figure 5.5 shows a Python stub that implements this interface and assigns proper values to a set of environmental variables.

While almost any programming language is capable of generating the application-specific PAL, we have chosen to use the Jinja2 template engine for the Contiki PAL generator (cf. Section 3.4.4). Figure 5.6 shows the Contiki template in excerpts. In this case, the Python stub from Figure 5.5 invokes the Jinja2 engine on that template,

```
numberof_threads = len(sys.argv) - 1
thread_ypfs      = map(lambda desc: desc.split(","), sys.argv[1:])
all_ypfs         = set(itertools.chain(*thread_ypfs))
pal_code         = sys.stdin.read()

sys.stdout.write(generate_pal())
```

Figure 5.5.: *Stub of a PAL generator:* Reading the PAL footprint from standard input and parsing the program arguments to retrieve the application footprint.

configuring it to scan for directives within C comments (slash-asterisk); as opposed to C++ style comments (slash-slash) which are considered as literal text. Overall, Jinja2 passes literal text unprocessed to the output, substitutes environmental variables, and processes simple directives like `if` and `for`. Instantiating the template from Figure 5.6 results in a Contiki PAL similar to the one from Figure 3.8.

The first two lines of the Contiki template include general OS-specific headers, while lines 4–7 include headers specific to the used yield point functions. After including all required headers, line 9 inserts the PAL footprint. Subsequently, lines 11–16 define the enumeration of yield point functions, while lines 18–25 define for each involved yield point function a data structure that holds the respective context (which is in contrast to Figure 3.8). Line 27–34 make use of these structures to create the `ThreadContext` structure, which is instantiated once for each thread in line 36. Next, Line 39–44 add the implementations of all involved yield point functions. As opposed to Figure 3.8, event handler code for the various yield point functions is grouped into a single function (lines 46–54), which is invoked individually by the event handler of each process that drives the execution of a T-code (lines 56–65). Line 66 and line 68 contain additional code which creates the required process structures and registers them with the autostart mechanism of Contiki. Please consult our source code distribution for details.

We think that a template engine like Jinja2 is a good choice to generate platform abstraction layers. As already mentioned in Section 3.4.3, a PAL generator script for platforms like TinyOS is, however, more involved. We have therefore chosen the previously described generic input-output interface via external program described to allow for such cases.

5.1.2.3. Optimizations

The main advantage of our compiler-assisted approach to cooperative threads is the efficiency of the generated code. Future work might therefore focus on adding additional optimization passes to the compiler pipeline. This section addresses this concern.

In general, there are two types of optimization passes: intra-procedural and inter-procedural ones. The first type of optimization performs optimizations on single critical functions while the second type performs optimizations on thread execution functions, which contain multiple critical functions. Consequently, there are two different points of the compiler pipeline where such optimizations should take place.

In order to make use of Hoopl [108], a rich implementation of the Lerner-Grove-Chambers algorithm for interleaved analysis and transformation [75], optimizations should operate on the IR, i.e., be functions from and to `Function`. Intra-procedural optimizations can easily be inserted into the pipeline after the `optimize` function. Inter-procedural optimizations, in contrast, could be performed right before the `threads` function or might have to be weaved into it right before turning the T-code in IR to E-code

```
#include "contiki.h"
// additional include directives

/*{ if "sleep" in all_ypfs }*/
#include "clock.h"
/*{ endif }*/
// additional conditional include directives

/* pal_code */

typedef enum {
    YPF_none = 0,
/*{ for ypf in all_ypfs }*/
    YPF_/*ypf*/,
/*{ endfor }*/
} YPF;

/*{ for ypf in all_ypfs }*/
typedef struct {
    tframe_/*ypf*/_t* frame;
/*{ if ypf == "sleep" }*/
    struct etimer et;
/*{ endif }*/
} ctx_/*ypf*/_t;
/*{ endfor }*/

typedef struct {
    union {
/*{ for ypf in all_ypfs }*/
    ctx_/*ypf*/_t /*ypf*/;
/*{ endfor }*/
    } ctx;
    YPF ypf;
} ThreadContext;

ThreadContext threads[/*numberof_threads*/];
ThreadContext* thread;

/*{ if "sleep" in all_ypfs }*/
void sleep(tframe_tc_sleep_t* frame) {
  // implementation of the sleep yield point function
}
/*{ endif }*/
// additional yield point functions

static void* event_handler(process_event_t ev, process_data_t data) {
    if (0) { }
/*{ if "sleep" in all_ypfs }*/
    else if (thread->ypf == YPF_sleep) {
      // handler for the sleep yield point function
    }
/*{ endif }*/
  // additional handlers for other yield point functions
}

/*{ for ypfs in thread_ypfs }*/
void thread_/*loop.index0*/(void*);
static char event_handler_thread/*loop.index0*/(
  struct pt* process_pt, process_event_t ev, process_data_t data) {
    thread = threads + /*loop.index0*/;
    // implementation of the PAL event handler using the generic event_handler function

    thread_/*loop.index0*/(continuation);
    return PT_YIELDED;
}
// generate process structure
/*{ endfor }*/
// generate code to automatically start all processes
```

Figure 5.6.: *The Contiki PAL generator template:* Jinja directives are escaped by C comments. The variables have been previously assigned by Figure 5.5.

in ASR. One should, however, carefully check if the intended optimization is already performed by the E-code compiler, because what is an inter-procedural optimization for Ocram might in fact be an intra-procedural optimization for the E-code compiler.

Section 6.5 names a few optimizations of both types that we envision to be worthwhile to implement because we expect them to increase the efficiency of E-code. Optimizing auxiliary functions in Ocram makes little sense because the E-code compiler is likely to already perform these optimizations. This is in contrast to critical functions, in which case the E-code compiler might not be able to compensate for missed optimization opportunities, as Section 6.5 also explains.

5.1.3. Implementation

This section highlights two interesting problems of the compiler's implementation and explains our solutions. Section 5.1.3.1 is concerned with traversing abstract syntax trees without manually writing repetitive traversal code. And Section 5.1.3.2 addresses our modifications to the printer to enable the collection of the printer log.

5.1.3.1. Traversal of Syntax Trees

Various steps of the compiler pipeline involve traversing a syntax tree either to query information or to transform the tree. Each of these traversals involve a lot of "boilerplate code", i.e., repetitive code fragments which are technically required but reflect little application-specific knowledge, if any.

In our case, the boilerplate code necessary to perform a top-down query of an abstract syntax trees, for example, is to provide one function for each AST data type (cf. Appendix A). Such a function would take a set of query functions, a log and an AST node of the respective type. It would then apply any matching query function to the node, merge the results into the log, call the respective function for each of the children of the node in order, passing the query functions and the current log in each case, and continuously update the log by merging the respective results and returning it finally. Similarly, transforming a syntax tree involves functions that unpack a node, transform its children in order and rebuild a new node with potentially new children. Usually, the extent of boilerplate code compared to "real" code, which actually implements the query or transformation logic, is huge. Worse, each kind of transformation or query tends to require its own set of boilerplate.

In an object-oriented environment like Java this calls for the visitor pattern, and in fact our first approach to this problem was to write a generic visitor-like framework to traverse abstract syntax trees. The problem with this however is that the framework and the traversal functions either have to be generic enough to accommodate all kinds of required query and transformation operation or there has to be a distinct set of traversal functions for each category of query and transformation operations. Unfortunately, both possibilities are not satisfying. The first one imposes its full complexity even to the simplest query and transformation operations, making them complicated and cumbersome to implement. The second way, on the other hand, requires to re-implement similar traversal code again and again, adding a lot of redundancy and opportunities for faults to the code.

In [71], Lämmel and Peyton Jones investigated this problem in the context of Haskell, resulting in a library for "generic programming" called Data.Generics. The key idea is to separate concerns by implementing the traversal of mutual recursive data types once in

```
data Error = MainFunction | NestedFunction

global_constraints :: CTranslationUnit a -> [Error]
global_constraints = everything (++) (mkQ [] scanExtDecl `extQ` scanBlockItem)

scanExtDecl :: CExternalDeclaration -> [Error]
scanExtDecl (CFDefExt fd)
  | name fd == "main" = [MainFunction]
  | otherwise         = []
scanExtDecl _         = []

scanBlockItem :: CCompoundBlockItem a -> [Error]
scanBlockItem (CNestedFunDef _) = [NestedFunction]
scanBlockItem _                 = []
```

Figure 5.7.: *Scrap your boilerplate:* Scanning for global constraints.

a generic way and to provide "type extensions" and "generic traversal combinators" to make use of this. A generic traversal combinator implements a specific traversal strategy for a given tree of mutually recursive data types. To this end, it needs to know how such a tree can in fact be traversed. This is encapsulated in the `Data` type class which must be implemented by all involved data types. The traversal combinator additionally needs a query or transformation function which is applied to the nodes of the tree and implements the actual logic of the traversal.

As different tree nodes have different types, the query or transformation function must be generic in some way. This is where the type extension functions come into play. A type extension function takes a query or transformation function that is specific to one data type and turns it into a generic function that takes an arbitrary type. The generic function wraps the specific function, calling it if the real type of its argument happens to match with the specific type, and performing a void operation otherwise. To facilitate this, it is required that both the specific input type and the arbitrary output type implement the `Typeable` type class, so that casting the types into each other is possible. Such a generic function is suitable to be used by a generic traversal combinator.

While conceptually the `Typeable` and `Data` instantiations have to be implemented for a given set of mutual recursive data types, the Glasgow Haskell Compiler (GHC) [85] can automatically generate them via the `DeriveDataTypeable` language extension. The Language.C library, which we make extensive use of in the prototypes, utilizes this feature by annotating all AST data type definitions with a `deriving (Data, Typeable)` clause. This makes GHC generate all the boilerplate code, thus drastically simplifying the implementation of query and transformation algorithms.

For example, Figure 5.7 shows a simplified implementation of the function `global_constraints`. Given a translation unit, the function returns a possibly empty list of errors, which, for the purpose of this example, can either be the presence of a main function or the definition of a nested function. This is a query operation and we use the generic traversal combinator `everything` to perform a top-down, left-to-right traversal. The first argument to `everything` is a function that is supposed to be used to merge the individual query results. In this example this is the list concatenation function ++.

The first argument of the type extension function `mkQ` is a default value that is used to perform possible void operations. Obviously, the empty list is the proper value for this. The second argument of `mkQ` is a specific query function. In our example, we actually want to use two distinct, specific query functions. We therefore use the `extQ` combinator which takes care that the void operation is only performed if neither of the two specific functions is applicable. Overall, `global_constraints` is defined in point-free style, i.e., the function is defined in terms of other functions without mentioning the function argument.

The individual query functions can focus on their respective task, as the generics framework makes sure to call them whenever a respective value is encountered during traversal. For example, if an external definition is a function definition and the name of that function is "main", then this is obviously an error which is returned in a list (line 8). If the name is something else (line 9) or the external definition is not a function definition (line 10), then there is no error, i.e., the returned error list is empty. Similarly, if a compound block item is a definition of a nested function, then this is reported as an error as well (line 13). If, in contrast, the compound block item is something else, then no error is reported (line 14).

This example illustrates two things. First, Haskell's pattern matching capabilities ease the implementation of the Ocram compiler. And second, Data.Generics makes it very simple to traverse trees of mutually recursive data types. This is particularly true in combination with GHC which generates the boilerplate code automatically. We make use of the generic programming technique in various places of the compiler and the debugger.

5.1.3.2. Printer Log

As explained in Section 4.2.2, the printer stage of the compiler pipeline memorizes for each AST node that is attributed by an enriched node info (ENI) where in the output document that node ended up being rendered to. While we use the pretty printer of the Language.C library, we had to extend it to support collecting such a log. This section describes how we did that.[5]

The pretty printer of the Language.C library itself depends on a second library, Text.PrettyPrint, whose design and algorithms have been published to provide an example for the capabilities of functional programming to facilitate software reuse [60]. The basic idea is to separate the concern of structuring a document from the algorithms that render documents in a nice way. Then, a client of the library can focus on the former aspect while the library itself takes care of the latter. To this end, Text.PrettyPrint provides a `Doc` data type which represents a document. A document is a recursive data structure, forming a hierarchical document tree. Various helper functions turn basic values like text and numbers into leaf documents, while various combinator functions layout partial documents to form a surrounding parent document. The authors of the Language.C library on their part provide functions that turn AST nodes into corresponding documents by using these helper and combinator functions. The root document of an AST node such as a `CTranslationUnit` is then passed to the `render` function that turns the `Doc` into a `String`. The algorithms behind `render` try to respect the layout information of the document to turn it into a nicely formatted string using one of currently four pre-defined strategies.

To support logging we had to extend both Text.PrettyPrint and Language.C. Concerning the former, we replaced the `Doc` data type with a polymorphic `DocL` data type which expects a type parameter that implements the `Monoid` type class (cf. Section 2.3.2). The idea behind this is to separate the concern of creating an empty log (`mzero`) and adding new values to it (`mappend`) from the actual implementation of that log. A trivial monoid is given by the empty set, represented in Haskell by the unit expression `()`. We make use of

[5]The basic idea originates from Simon Meier, Institute of Information Security, ETH Zurich.

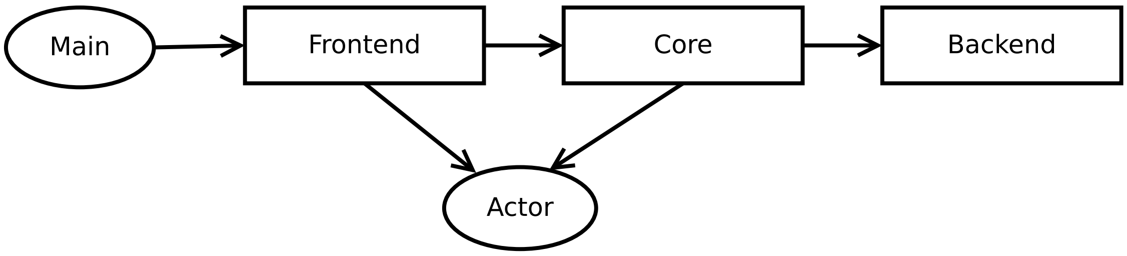


Figure 5.8.: *The Ruab modules:* We only show top-level modules and omit dependencies to infrastructure modules.

this to maintain downwards compatibility by defining `type Doc = DocL ()`. Usually, however, the log is a list of some user-defined values, while `mzero = []` and `mappend = (++)`.

A `DocL` is just like a `Doc` except that it can be augmented with a `Logger`. A logger is a function that takes a position and returns a monoid value, i.e., a new log entry. Whenever a document is rendered to the output string and has an attached logger, that logger is invoked with the current position of the output string. The printer log starts with `mzero` and is accumulated along the way by adding the values returned by the individual loggers via `mappend`. The result of the render function is the output string as before plus the final log.

Next, we changed the pretty print functions of the Language.C library that turn AST nodes into documents. For our purpose, `PrinterLog` is a proper log type as it is a list of pairs consisting of an ENI and an E-code row (cf. Section 4.2.2.2). Whenever an AST node is attributed by an ENI, its document, which is created as before, is augmented with a logger. That logger receives the current position and returns a `PrinterLog` with one element, the ENI of the AST node paired with the E-code row.

Overall, this is a great example of the expressiveness, composability and flexibility of functional programming languages. Contributing our changes to these libraries is currently ongoing.

5.2. Ruab: T-code Debugger

This section follows the structure of Section 5.1 and covers three aspects of Ruab, our T-code debugger prototype. First, Section 5.2.1 shows the overall software architecture. Then, Section 5.2.2 documents key interfaces of Ruab. Finally, Section 5.2.3 highlights a few interesting implementation issues.

5.2.1. Architecture

This section provides an overview of the architecture of Ruab. Table 5.2 lists all Haskell modules that constitute the T-code debugger, and Figure 5.8 shows an excerpt of the dependency graph of these modules. Just like in Section 5.1.1, we will only provide a few comments on the modules whose concepts are not covered elsewhere in this thesis. Please consult our source code distribution for details.

The various sub-modules of the GDB back-end implement the client-side of the GDB Machine Interface. This involves data types, parsers and printers for the various

module	section	description
Actor	5.2.3.2	functional actors
Backend	4.3	back-end modules
GDB	5.2.3.3	GDB back-end module
Commands	n/a	GDB/MI commands
IO	5.2.3.3	controlling a GDB instance
Representation	n/a	GDB/MI data types, parser and utility functions
Responses	n/a	GDB/MI responses
Core	4.4	the Core module
Internal	4.4.3	evaluating expressions
Front-end	5.2.3.1	a GTK-based front-end
Infos	5.2.3.1	visualizing breakpoints, threads, etc.
Main	n/a	integration
Options	n/a	process command line options
Util	n/a	utility functions

Table 5.2.: *The Ruab architecture:* The responsibilities of the various modules and submodules in alphabetical order.

commands, responses and notifications. We used the hgdbmi library[6] as a starting point and extended it substantially. Contributing our implementation back to the hgdbmi project is currently ongoing.

The Main module instantiates the front-end module, which is responsible to setup the Core module, which in turn starts the back-end module. Overall, this resembles a model-view-controller architecture.

The Options module defines and processes command line options for the Ruab executable. Possible options are the file path to the target binary, the file path to the debug information file, and an optional file path for a log of GDB/MI communication.

Finally, the Util module provides various utility functions used throughout the project. Like the Ocram module, the Ruab modules are also paired with unit and integration tests.

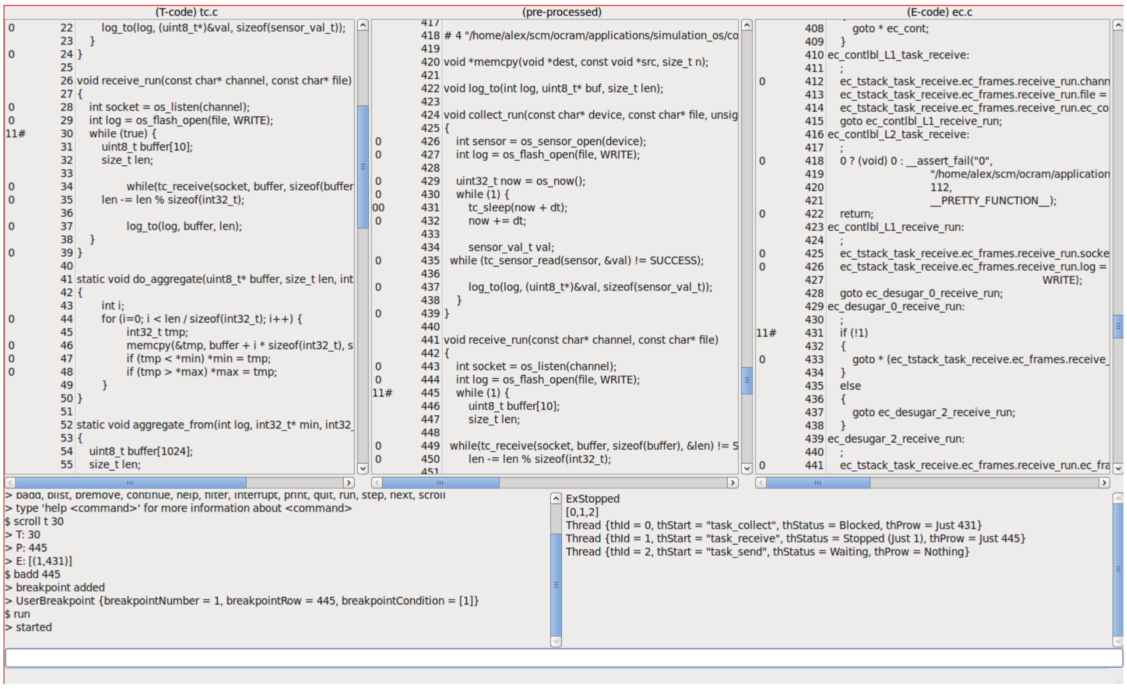
5.2.2. Interfaces

This section explains the interface between Ocram and Ruab. This is the only interesting interface of Ruab that has not been covered previously.

Interfacing Ocram and Ruab is implemented by the `Ocram.Ruab` module (cf. Section 5.1.1). This module is shared by both projects and includes three things. First, it specifies debug information data types. Second, it defines the storage format of the debug information via marshalling and unmarshalling functions. And third, it provides functions to translate T-, P- and E-code rows into each other, because these functions are needed by both Ocram and Ruab.

The current storage format is based on the *JavaScript Object Notation* (JSON) [65] and is implemented via the Text.JSON library[7]. In principle, any other storage format

[6] `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/hgdbmi`
[7] `http://hackage.haskell.org/package/json`



(T-code) tc.c

```
0      22       log_to(log, (uint8_t*)&val, sizeof(sensor_val_t));
       23    }
0      24 }
       25
       26 void receive_run(const char* channel, const char* file)
       27 {
0      28    int socket = os_listen(channel);
0      29    int log = os_flash_open(file, WRITE);
11#    30    while (true) {
       31       uint8_t buffer[10];
       32       size_t len;
       33
0      34             while(tc_receive(socket, buffer, sizeof(buffer
0      35       len -= len % sizeof(int32_t);
       36
0      37             log_to(log, buffer, len);
       38    }
0      39 }
       40
       41 static void do_aggregate(uint8_t* buffer, size_t len, int
       42 {
       43       int i;
0      44       for (i=0; i < len / sizeof(int32_t); i++) {
       45             int32_t tmp;
0      46             memcpy(&tmp, buffer + i * sizeof(int32_t), s
0      47             if (tmp < *min) *min = tmp;
0      48             if (tmp > *max) *max = tmp;
       49       }
       50 }
       51
       52 static void aggregate_from(int log, int32_t* min, int32_
       53 {
       54    uint8_t buffer[1024];
       55    size_t len;
```

(pre-processed)

```
       417
       418 # 4 "/home/alex/scm/ocram/applications/simulation_os/co
       419
       420 void *memcpy(void *dest, const void *src, size_t n);
       421
       422 void log_to(int log, uint8_t* buf, size_t len);
       423
       424 void collect_run(const char* device, const char* file, unsig
       425 {
0      426    int sensor = os_sensor_open(device);
0      427    int log = os_flash_open(file, WRITE);
       428
0      429    uint32_t now = os_now();
0      430    while (1) {
00     431       tc_sleep(now + dt);
0      432       now += dt;
       433
       434       sensor_val_t val;
0      435   while (tc_sensor_read(sensor, &val) != SUCCESS);
       436
0      437       log_to(log, (uint8_t*)&val, sizeof(sensor_val_t));
       438    }
0      439 }
       440
       441 void receive_run(const char* channel, const char* file)
       442 {
0      443    int socket = os_listen(channel);
0      444    int log = os_flash_open(file, WRITE);
11#    445    while (1) {
       446       uint8_t buffer[10];
       447       size_t len;
       448
0      449   while(tc_receive(socket, buffer, sizeof(buffer), &len) != S
0      450       len -= len % sizeof(int32_t);
       451
```

(E-code) ec.c

```
       408       goto * ec_cont;
       409    }
       410 ec_contlbl_L1_task_receive:
       411    ;
0      412    ec_tstack_task_receive.ec_frames.receive_run.chann
       413    ec_tstack_task_receive.ec_frames.receive_run.file =
       414    ec_tstack_task_receive.ec_frames.receive_run.ec_co
       415    goto ec_contlbl_L1_receive_run;
       416 ec_contlbl_L2_task_receive:
       417    ;
0      418    0 ? (void) 0 : __assert_fail("0",
       419                        "/home/alex/scm/ocram/application
       420                        112,
       421                        __PRETTY_FUNCTION__);
0      422    return;
       423 ec_contlbl_L1_receive_run:
       424    ;
0      425    ec_tstack_task_receive.ec_frames.receive_run.socke
0      426    ec_tstack_task_receive.ec_frames.receive_run.log =
       427                                                  WRITE);
       428    goto ec_desugar_0_receive_run;
       429 ec_desugar_0_receive_run:
       430    ;
11#    431    if (!1)
       432    {
0      433       goto * (ec_tstack_task_receive.ec_frames.receive_
       434    }
       435    else
       436    {
       437       goto ec_desugar_2_receive_run;
       438    }
       439 ec_desugar_2_receive_run:
       440    ;
0      441    ec_tstack_task_receive.ec_frames.receive_run.ec_fra
```

```
> badd, blist, bremove, continue, help, filter, interrupt, print, quit, run, step, next, scroll
> type 'help <command>' for more information about <command>
$ scroll t 30
> T: 30
> P: 445
> E: [(1,431)]
$ badd 445
> breakpoint added
> UserBreakpoint {breakpointNumber = 1, breakpointRow = 445, breakpointCondition = [1]}
$ run
> started
```

```
ExStopped
[0,1,2]
Thread {thId = 0, thStart = "task_collect", thStatus = Blocked, thProw = Just 431}
Thread {thId = 1, thStart = "task_receive", thStatus = Stopped (Just 1), thProw = Just 445}
Thread {thId = 2, thStart = "task_send", thStatus = Waiting, thProw = Nothing}
```

Figure 5.9.: *The Ruab GUI:* The right-most view port shows E-code only to facilitate the development of Ruab.

could be used. The only thing to consider is that it has to be able to represent C source code, as the P-code of the application is stored along with the debug information.

Since both T-code and E-code applications are only referenced via a file path, additional measures are required to guarantee that Ruab uses the same version of these files as Ocram did when generating the debug information file. The current implementation therefore augments each file path with the MD5 sum [112] of the contents of that file.

5.2.3. Implementation

This section highlights a few interesting implementation aspects of Ruab. Section 5.2.3.1 presents our GTK-based front-end. Then, Section 5.2.3.2 explains that the Ruab architecture is based on actors. Finally, Section 5.2.3.3 documents the implementation of the GDB back-end module. It particularly covers our implementation of the GDB Machine Interface.

5.2.3.1. GTK front-end

The only front-end module that we implemented is based on GTK+[8], “a multi-platform toolkit for creating graphical user interfaces”. It employs the Gtk2Hs library[9], a GUI library for Haskell based on GTK+. The GUI is designed with Glade[10], “a rapid application development tool [for the] development of user interfaces for the GTK+ toolkit”.

Figure 5.9 shows a screenshot of the GTK-based Ruab front-end. The three view ports show T-code, P-code, and E-code. The latter is only added to facilitate the development of Ruab. An official Ruab release should hide this view port, because hiding E-code details

[8]`http://www.gtk.org/`
[9]`http://projects.haskell.org/gtk2hs/`
[10]`http://glade.gnome.org/`

is exactly the purpose of the T-code debugger. The three view ports are synchronized to show corresponding source code rows. The lower-left window shows a history of user commands and corresponding results while the lower-right window shows the status of the debugger and the application threads. User commands can be entered via the input field at the bottom.

In the exemplified session, the DCA application (cf. Section 6.1.2) is loaded in the T-code debugger. The user issued the `scroll` command to sync the view ports to the T-code row number 30. This command is internal to the front-end and returns the translated P-code and E-code rows. These rows are marked with a hash symbol in the corresponding view ports. In the E-code case, the result is a list, because in case of re-entrant functions there is one row per thread (cf. Section 4.2.2.2). Instead of showing all E-code rows that correspond to a given P-code row and thread, only the smallest one is returned, as it contains the first corresponding E-code statement.

The command `badd` has been propagated to the Core to add a new user breakpoint at the given P-code row. The columns of the three view ports that contain mostly zeros mark each row that can be augmented with a breakpoint. The T-code row 30 and the corresponding P-code an E-code rows are marked with a 1 in that column, indicating the number of the previously added breakpoint.

The subsequent `run` command has started the debugger such that the previously added breakpoint was hit next. The status window therefore shows `ExStopped`, which means that the debugger is in the `stopped` state. The next line shows the status of the global thread filter. Since it has not been set in the given sessions, it still defaults to include all threads.

The next three lines show the status of the three T-code threads. Each of them names the thread ID, the thread start function, the state of the thread and its current row. The first thread is currently blocked at P-code row 431. The second thread has stopped at breakpoint 1, P-code row 445. And the third thread is still waiting, i.e., it has not been started yet.

Obviously, the Ruab GUI is not very user friendly. Nevertheless, it covers all concepts of the T-code debugger.

5.2.3.2. Actors

Haskell is a pure programming language. This means that side effect have to be performed explicitly. Obviously, the Core module involves a lot of user interaction (i.e., I/O) and internal state. Additionally, the GTK front-end and the GDB back-end both run their own mutually independent threads. In order to manage both of these aspects, we have decided to base Ruab's software architecture on actors [20].

An *actor* runs its own thread and encapsulates a state. It receives update functions that alter that state. This design is borrowed from the actors of the standard library of Clojure[11], a Lisp dialect for the Java Virtual Machine. An update function may perform additional I/O operations. It can particularly send other update functions to different actors. This is the basis of the communication between actors.

Figure 5.10 shows that the current implementation of Ruab employs three actors. The first one implements the state machine of the Core module. It receives commands from the front-end and notifications from the back-end, and it sends thread status updates to the front-end.

[11]`http://clojure.org`

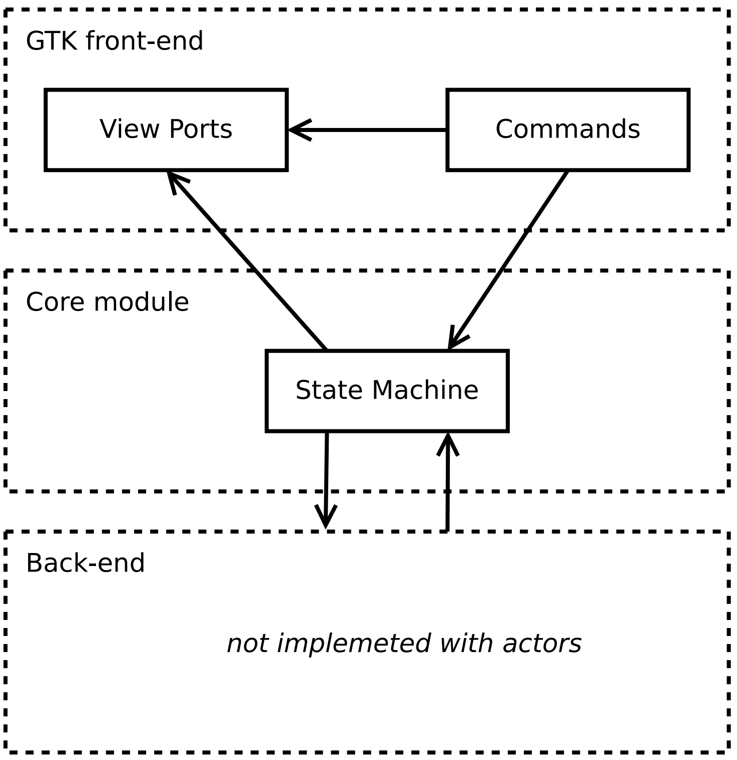


Figure 5.10.: *Actor-based architecture of Ruab:* Actors encapsulate a state and send and receive update functions.

The second actor implements the command interpreter in the GTK front-end. The command interpreter processes command inputs from the user, and it is stateful because it supports basic command history. When a valid command is entered, the actor sends that command either to the Core module or, in case of an internal command, to the third actor. This actor implements the output side of the graphical user interface. It manages the view ports for the application source code and displays the current row and all user breakpoints. The input to this actor are thread status updates from the Core module as well as internal commands.

While the actor communication crosses module interfaces, the expressiveness of Haskell allows for keeping these interfaces agnostic to this. In fact, a future Core module could cease to use an actor and neither the front-end interface nor any of the front-end implementations would have to be changed. A good example that makes use of this flexibility is the GDB back-end that is presented in the next section. It does not use actors only because we don't want it to depend on them, as we are planning to publish our GDB/MI implementation separately. The design of the back-end interface and the implementation of the Core module are, however, not affected by this decision.

5.2.3.3. GDB Machine Interface

The *GDB Machine Interface* (GDB/MI) is a machine-readable variant of GDB's text-based command line interface (Section 27 of [129]). It is intended to be used by debugger front-ends such as Eclipse and is based on sending and receiving strings via standard input and standard output.

Our GDB module forks a GDB instance in the background and connects to its standard input and output via UNIX pipes. It also provides a synchronous function to send commands and receive responses, as well as an asynchronous interface to receive notifications.

Internally, a single thread reads from a queue of pending commands, renders each of them in turn to GDB/MI syntax, and sends it to the standard input of the GDB. While there is exactly one response for each issued command, notifications can occur at any point in time. Possible notifications include a breakpoint that has been hit or the termination of the inferior. This requires a second thread that continuously reads the standard output of the GDB and parses the GDB/MI syntax to obtain either a response or a notification.

When a client issues a new command a new synchronising variable (`MVar`) is created. The client call blocks on this variable until the corresponding response is available. Notifications, in contrast, are delivered to the client via registered callback functions. Because these callback functions might block, which would freeze the GDB module, they are invoked in a separate thread. This is cheap, because the Haskell run-time systems provides lightweight user-land threads.

When the debugger is shut down, the debug instance is terminated via the GDB/MI quit command and the two internal threads are killed. Also, each of them is joined, leaving the GDB module in a clean state. Subsequent commands are rejected from then on.

5.3. Summary

In this chapter we provided a basic documentation of our prototypes, Ocram and Ruab. In each case we presented the architecture, specified major interfaces, and discussed a few non-trivial implementation problems. Overall, the contribution of this chapter was to make our source code distribution accessible and to enable future researchers and engineers to extend or port the prototypes and to perform their own experiments.

6. Evaluation

Chapter 5 introduced our prototypical implementations of a T-code compiler (cf. Chapter 3) and a T-code debugger (cf. Chapter 4). In this chapter, we will use them to evaluate both the correctness as well as the performance of our compiler-assisted approach to thread abstractions for resource-constrained systems. Section 6.1 documents the setup of the conducted experiments. To make sure that the results of the experiments actually represent the properties we are interested in, a number of verification steps are taken, as Section 6.2 explains. Section 6.3 follows with a description of the measurements taken during the experiments and Section 6.4 presents the results of those. In Section 6.5 we discuss these results and approach various possibilities to improve them further. All of the above sections cover the evaluation of the T-code to E-code translation and our T-code compiler prototype. Section 6.6, in contrast, covers a less extensive evaluation of the T-code debugger prototype.

6.1. Experiments

To evaluate our compiler-assisted thread abstraction, we have chosen a set of three case studies. Each of them follows a real WSN application archetype, so that the evaluation is realistic (Section 6.1.2). Furthermore, we tried to cover a representative range of application types, programming concepts and concurrency patterns as summarized by Table 6.1. We implemented each of these case studies in three variants: 1) a native event-based version, 2) a thread-based version using a thread library, and 3) a T-code version using Ocram (Section 6.1.1). All nine resulting programs are written for Contiki [30] and we have executed them via COOJA/MSPsim [38], while an extra COOJA plugin collected various logs and measurements. Appendix C specifies the details of the setup and documents how to reproduce the experimental results.

6.1.1. Variants

For each case study application we first implemented a *native variant* (NAT) using the event-based paradigm. To this end, we either copied existing code or wrote an implementation following common programing patterns as encountered in the Contiki community. This implies using protothreads [31], which disguises that the run-time system is event-based and adds an overhead of two bytes per protothread. We argue, though, that this does not significantly bias the ground truth of our evaluation, which is the performance of a native event-based application. Furthermore, we get a nice comparison between our comprehensive thread abstraction and the allegedly most efficient, but limited thread abstraction of protothreads.

Compiler-assisted thread abstractions compete with run-time-based approaches and in Section 1.4 we have argued why we think that dedicated compilers can produce more efficient code. To verify this hypothesis, we have secondly written a *threaded variant* (TL) which is directly executed using a thread library. For this purpose, we have ported

	DCA	COAP	RPC
client side	n/a	yes	yes
server side	n/a	no	yes
application layer	yes	yes	no
protocol layer	no	yes	yes
inter-thread communication	yes	yes	yes
thread synchronization	no	yes	yes
re-entrance	no	yes	yes
producer-consumer	yes	no	no
thread pool	no	no	yes

Table 6.1.: *Properties of the case study applications*

the TinyThreads [88] thread library to Contiki, as it is the only available full-fledged thread library for cooperative threads in sensor networks. We kept the basic context switching code and the general scheduler architecture, but we had to adapt the details to Contiki's APIs. We also removed support for preemption and dynamic thread creation to avoid extra overhead for features that our abstraction does not provide. Overall, a single protothread executes the scheduler and all application-level threads. When an application invokes a blocking function, the corresponding Contiki operation is triggered and the context switches back to the scheduler. Likewise, when the corresponding event is delivered to the scheduler process, context switches back to the application and the blocking function completes.

Finally, the third variant is the *generated variant* (GEN) which is an E-code application that has been generated from a T-code application by Ocram. To enable a fair comparison, the three variants only differ from each other if the different programming models require so. They thus share common code via utility modules. Also we copied as many source code lines between the variants as possible. In particular, the two thread-based variants differ only in the way threads are declared and started and in the names, but not in the signatures of the blocking functions. As a single exception to this, the `receive` operation for TL needs to pass the network connection of interest because otherwise the single scheduler protothread cannot demultiplex incoming network packets.

6.1.2. Case Study Applications

Figure 6.1 shows simplified pseudo code for the three case study applications. The pseudo code resembles a thread-based programming style and we will use the corresponding terminology consistently to keep things simple in the following. When we use this terminology while referring to NAT, we mean the corresponding terms. For example, the term "blocking function" in the context of NAT addresses the corresponding asynchronous function that triggers the same operation. Such a corresponding thing always exists because, as we will see later, all three variants pass the same black box test, so all of them implement the same application in each case.

So, the three applications use the blocking functions `sleep`, `receive`, and `wait`, and all three of them can be interrupted via `notify`. Note that both reading sensor values and sending network packets is a synchronous operation on Contiki and we have decided against artificially turning them into asynchronous operations to avoid biasing the comparison with the native variant.

```
receiving:
    receive
    add values to ring buffer

collecting:
    sleep 23 second
    read from sensor
    add value to ring buffer

sending:
    sleep 127 seconds
    empty ring buffer
    aggregate values
    send results to parent
```

(a) DCA

```
transactions:
    sleep until next timeout
    if notified:
        add new transaction to queue
    else:
        send transaction
        double transaction timeout

receiver:
    receive
    cancel pending transaction
    otify blockwise transfer

client:
    sleep 10 seconds
    blockwise transfer PUT request
    blockwise transfer GET request

blockwise transfer:
    for each block:
        create transaction
        send transaction
        wait
```

(b) CoAP

```
server:
    receive
    if notified:
        send response
    else:
        notify available worker

worker [1-N]:
    wait
    handle call

handle call:
    if read fast sensor:
        read sensor
    if read slow sensor:
        sleep // emulate slow sensor
        read sensor
    if tell:
        send contained call to peer
        receive response
    notify server
```

(c) RPC

Figure 6.1.: *Pseudo code of the case study applications:* We use distinct formatting to mark thread start functions, *re-entrant critical functions*, and **blocking functions**.

The first case study is a typical *data collection and in-network aggregation* (DCA) application consisting of three tasks. Overall, the application reads values from the local sensor, receives values from its child node(s) and sends aggregated values to its parent node. As summarized in Table 6.1, this constitutes a consumer-producer pattern with inter-thread communication via a shared ring buffer, but no explicit thread synchronization and no re-entrant code. The major architecture of this case study can be found in many deployments, such as the PermaSense project [52]. In order to focus on the issues of the programming model, we implemented this application from scratch, leaving out advanced data aggregation algorithms, network packet framing, etc.

As we are interested in the performance of the software on a single mote, we do not simulate large networks. Instead, the simulation for this application only consists of three motes. One mote is running the respective variant of this application, while two other motes execute the child and parent application, respectively. The simulation ends as soon as the parent node receives the fifth packet. This choice is arbitrary but large enough to cover all code paths and possibly even out transient effects. For the simulations of the other two applications we have made similar choices, and overall all three applications have a similar total simulation length.

The second case study is a complete client-side implementation of the CoAP protocol [121] including block-wise transfer[1] [12], and a small application layer. The program consists of three tasks and overall the client repeatedly sends PUT and GET requests to the server. The PUT request sets the seed for a random resource on the server, while the GET request retrieves a possibly large sequence of characters from this resource. Both the value of the seed and the length of the character sequence are chosen randomly. As listed in Table 6.1, this application involves both explicit thread synchronization via `wait` and `notify` as well as re-entrant code. The native implementation for this study is taken from Contiki's CoAP implementation [70] and the simulation consists of one mote running the variants of the CoAP client and one mote running the CoAP server. The simulation ends as soon as the server receives the seventh PUT request.

The third case study is motivated by a programming framework for sensor networks [95] that offers so-called *tell actions*. A tell action is a one-to-many *remote procedure call* (RPC), which means that a node may instruct one or more other nodes to execute a potentially blocking operation. The tell action itself blocks until all nodes have finished executing the command. In a network in which multiple tell actions can occur at any time, each node should be able to handle multiple RPCs at the same time. This calls for a thread pool, a common concurrency pattern that can be found in many RPC systems such as CORBA [47]. Although there are some RPC frameworks for sensor networks [87, 146], none of them supports concurrent invocations. Thus, we implemented this framework from scratch.

In order to focus on the programming model again, we only support three basic remote calls that conceptually cover the whole spectrum of interest: 1) reading a value from a fast sensor such as a temperature sensor, 2) reading a value from a slow sensor, which involves some startup time and thus a blocking function on the callee's side, and 3) a tell operation, which involves delegating any of these three remote calls to a different node and thus also requires a blocking function on the callee's side.

As shown in Table 6.1, the application implements both the client and the server side of the RPC protocol and involves explicit thread synchronization and re-entrant functions. The simulation for this application consist of three motes: one client and two peers. The

[1]application-level payload fragmentation via stop-and-wait

first peer is the one we evaluate and it runs one of the three application variants, while the second peer always executes the native variant of this application. The client issues a tell action to the first peer, which delegates the included call to read from a slow sensor to the second peer. While the tell call is pending, the client sends an additional call to the first peer, having it reading from its fast sensor, so that both of its workers become busy at the same time. As soon as the reply of the tell action arrives at the client, it starts all over again and performs this loop four times.

6.2. Verification of the Experiments

The verification serves four purposes. First, we want to make sure that there are no software faults in the T-code compiler. Second, we want to guarantee that we measure only the effects of the different programming abstractions. Third, we want to test each variant of each application to rule out software faults. And last, we want to confirm the correctness of the transformation.

To check the T-code compiler for software faults we have written a large set of unit tests for each of the steps of the compiler pipeline. We have also written various integration tests for each of its modules as well as for the whole pipeline. We carefully tried to cover all cases we could think of and verified the generated code manually in each case. Overall, we think the software quality of Ocram is good enough to conduct these experiments in good conscience. However, tests inherently only provide selective verification, and with regard to the complexity of C99 and GNU C, it would not take us by surprise if Ocram missed some corner cases. We cover this possibility with the following additional verification steps.

We wrote a COOJA plugin that collects a log of `printf` [C99: 7.19.6.3] traces, which serves as an input to an application-specific verification script. This constitutes a black box test for each variant of each application, which covers the implementation of each and indirectly also Ocram's functionality due to GEN. The same plugin also collects a log of the observable behavior which is used to compare TL with GEN. As TL de facto constitutes the execution of the T-code application, this comparison verifies the correctness of the transformation and its implementation — given there is no software fault in the thread library that results in the exact same wrong behavior.

In fact, the comparison checks if the observable behaviors are equal, which is easy to implement, but stronger than what is required by the definition of equality. Time-dependent parameters such as the sleep duration for calls to `sleep` may, however, differ because they depend on when exactly the corresponding `get_time` function has been invoked. In these cases, we accept deviations up to six milliseconds and reject anything else. Note that this does not violate the definition of equality, as slightly earlier or later calls to `get_time` are possible for the T-code application, as cooperative threads give no timing guarantees.

In order to obtain an unbiased comparison of the variants, we used the same COOJA simulation for all of them, only exchanging the binary under test in each case. This means that spatial mote distribution, radio model, behavior of neighbor nodes, random seeds, etc. are constant. That is, the execution environment is deterministic and produces the same results repeatably. Additionally, we used a difference tool to make sure that the source code of the three variants only differs when the respective programming abstraction requires so.

6.3. Performance Measurements

As a first and simplest measurement, we counted the *lines of code* required to implement the application logic for each variant. Lines of code is in general not very significant, but when using identical code formatting rules, as we did in our evaluation, it provides a quick estimation of the expressiveness of the programming model. In order to focus on the effects of the different programming models, we did not include the shared utility modules into the counting. Similarly, we neither took GEN's PAL nor TL's scheduler into account, as this code needs to be written only once and can thus be regarded as being part of the operating system, which we did not count either.

Next, we compiled each application for the Tmote Sky platform using the MSP430 port of the GCC [79] version 3.2.3. The compilation process was performed by Contiki's build system with `SMALL = 1`, which amongst other things instructs the linker to remove unused functions. To obtain static measurements from the resulting ELF [134] binary we used `objdump` from GCC's binutils[2] and retrieved the size of the *text section* (i.e., the machine code itself), the size of the initialized *data section*, and the size of the uninitialized data section, also known as *bss section*.

Besides these static measurements, we were also interested in run-time properties. To obtain a precise count of *CPU cycles*, we modified the Contiki system[3] by introducing a variable called `process_hook` with type pointer to `void`. Right before invoking a process, the scheduler writes the address of the descriptor of the particular process to `process_hook`. And right after the process returns control, `process_hook` is set to `NULL`. Our plugin can thus install a breakpoint for modifications to `process_hook`, which enables it to precisely sum up CPU cycles for each process individually. All interrupt functions and the code that logs the observable behavior signal their invocation via `process_hook` as well. The plugin can thus remove these CPU cycles from the current process' account, and by considering the cycles for the prologue and the epilogue of the interrupt handler functions and the cycles for the write operations to `process_hook`, it measures the exact number of CPU cycles per process.

For NAT, we counted the CPU cycles of all processes that run an application task, leaving out any OS processes. For TL, we counted the CPU cycles of the single scheduler process only, as it executes all application threads. And finally, for GEN, we counted the CPU cycles of all Contiki processes that execute an application thread (cf. Section 3.4). Overall, the counted CPU cycles cover the same application functionality in each case.

In order to measure the *maximum stack consumption*, our plugin installs a breakpoint for updates to the *stack pointer register* (SP). For NAT and GEN, tracking the maximum SP value and subtracting it from the start address of the stack is sufficient. For TL, we also need to take the stacks of the application threads into account, though. Our plugin does this accurately and thus obtains the precise amount of bytes required for each stack. We used these values to set the size of the application stacks, thus reducing the size of the bss section as much as possible and enabling a fair comparison. As interrupts happen nondeterministically, a single simulation run might not catch the worst case of interrupt function invocations, though. Thus, we added a safety margin of 20 bytes to each stack and so far no stack overflows occurred during our measurements, which of course is also monitored by the plugin.

One thing that we do not measure is the energy consumption as it is largely driven by

[2]`http://www.gnu.org/software/binutils/`

[3]The patches are included in our source distribution.

radio communication which is part of the observable behavior. As the transformation preserves the observable behavior, it does not add to the radio-driven energy consumption.

6.4. Results

A major observation of the evaluation is that the results are deterministic. Thus we can directly interpret these values without any additional statistics methods.

Figure 6.2a shows that in order to implement the same application, T-code requires 8 % to 17 % less lines of code than a native Contiki implementation. As already mentioned in the previous section, this measurement does not cover shared utility modules, which contain additional 2000 lines of code for CoAP and 300 for RPC. As protothreads do already help in compacting event-based code, this measurement fails to provide the ground truth of a real event-based application. But as “with protothreads the number of lines of code was reduced by one third” [31], we can estimate that a T-code application requires up to 45 % less lines of code than an equivalent event-based application. Although this result is not precise, it still supports our initial motivation for this work: synchronous operations and sequential computation provide an easier programming model than asynchronous operations and event handler functions. The TL variant is close to GEN but higher because it provides the same programming model as GEN, but requires extra lines to define the application stacks and to start the threads.

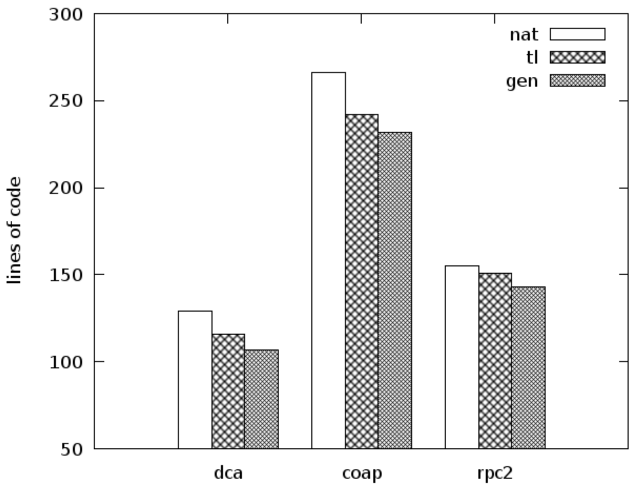
6.4.1. Memory Resources

As there is no free lunch, we expect our thread abstraction to also have some costs. First of all, we are interested in the overall memory consumption because RAM is very limited on sensor network devices. In this regard, Figure 6.2b shows the size of the data and the bss section along with the maximum stack size for each variant of each application. First, we can see that all three variants have roughly the same amount of initialized data and a large common block of bss memory. This is because each variant uses the same operating system that adds its string constants, network stack buffers, etc.

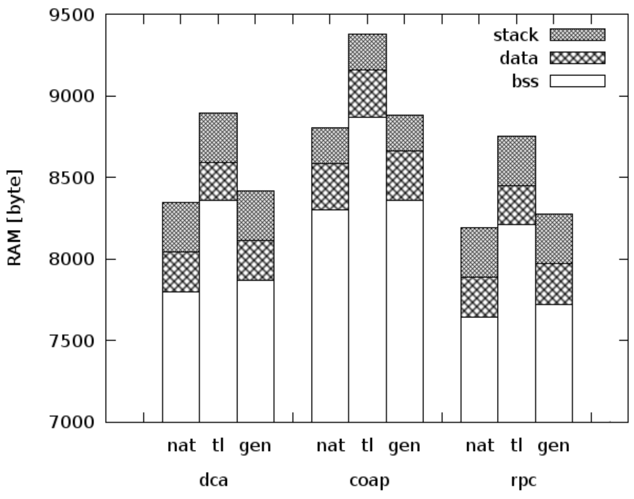
Additionally, we can see that all three variants have roughly equal maximum stack sizes because none of them uses the hardware stack a lot: Protothreads use function-static variables, TL uses the stacks of the application threads for local variables, and GEN uses its T-stacks. As a consequence, we can see TL and GEN having larger bss segments.

The interesting thing is that TL’s overhead is significantly higher which reflects our initial motivation for this work: comprehensive run-time-based thread abstractions are not resource-efficient (cf. Section 2.1.3). A second interesting observation is that both GEN’s overhead of the bss section and its overhead of the total amount of required RAM is approximately 1 % compared to NAT.

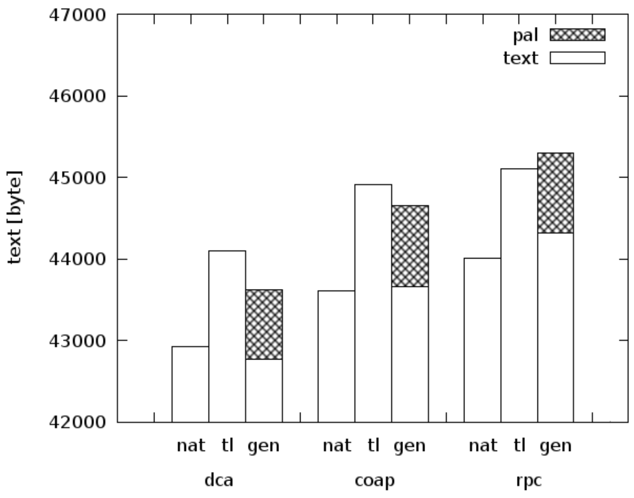
Another limited resource of sensor network devices is ROM space, which means that we need to compare the binary size of the variants. Figure 6.2c thus shows the size of the text sections and, in case of the GEN variant, it also distinguishes between code resulting from the application layer and code added by the PAL. First, we see the overhead of TL’s scheduler as expected. Similarly, we see the overhead of GEN’s PAL. But we also see that the generated code itself, i.e., the E-code application not including the PAL, is almost the same size as the code of the NAT variant. And including the PAL, the overhead is below 3 %.



(a) lines of code

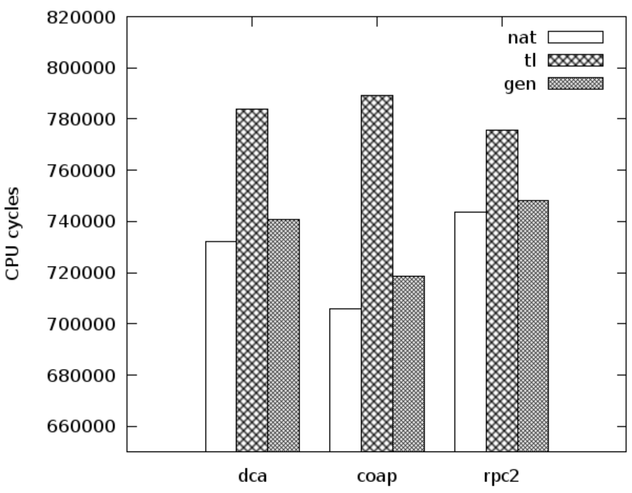


(b) RAM

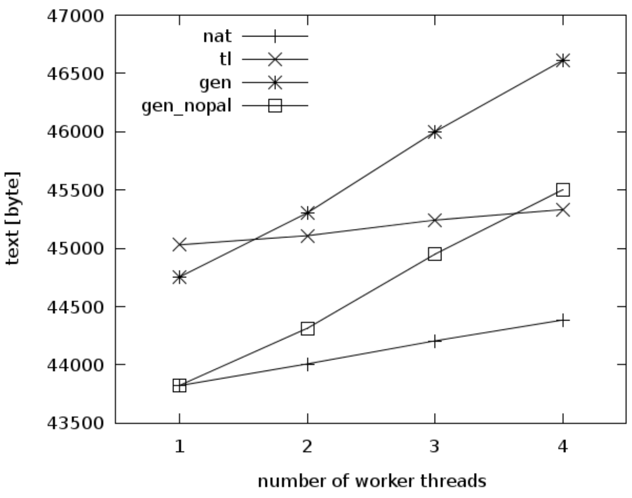


(c) text

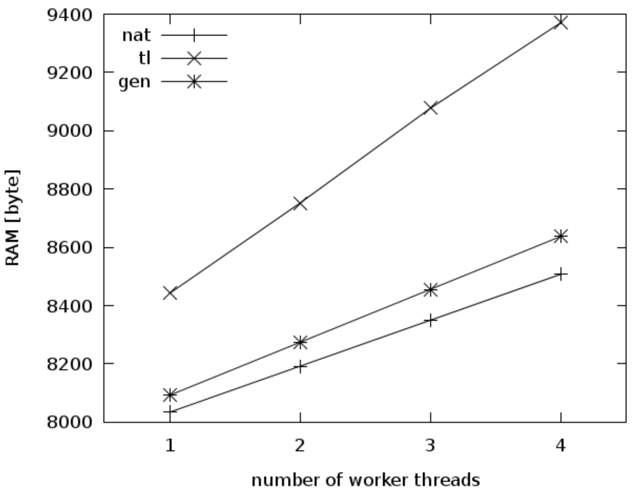
Figure 6.2.: *Evaluation results:* Resource consumption of various resources per variant and application.



(a) CPU cycles



(b) text per thread



(c) RAM per thread

Figure 6.3.: *Evaluation results:* Resources consumption of RPC application per variant versus number of threads.

Overall, the results show that Ocram provides the comfort of threads for just a small amount of extra RAM and ROM space.

6.4.2. CPU Cycles

Next, we want to know if the generated code involves more computation than a native implementation because keeping the CPU busy prevents the device from going into a low-power idle state. Referring to this, Figure 6.3a shows the CPU cycle count for each variant of each application. The absolute range of the values depends on the duration of the simulation of each application and thus provides little insight. But what we do see is that TL's scheduler adds up to 12 % of CPU cycles compared to the NAT variant. And we see that despite the additional PAL, GEN's number of CPU cycles is only approximately 2 % higher compared to NAT.

Although we do not know the exact division between the PAL and the application code, regarding the extra work performed by the PAL, this result suggests that the E-code actually requires less CPU cycles than the native implementation. An explanation for this is that all critical functions are inlined in the E-code, thus saving extra cycles for function prologue and epilogues. But of course, this comes at the cost of a larger binary size in case of re-entrant functions.

6.4.3. Per-Thread Consumption

To analyse the costs of inlining critical functions, we varied the number of worker threads of the RPC application from one to four and performed the measurements for each configuration. In each case, we adapted the client mote to send the right amount of concurrent remotely blocking RPC calls to have all worker threads busy at the same time. The previous figures showed results for the RPC application with two worker threads and using more than four workers already exceeds the limited resources of the Tmote Sky.

Figure 6.3b shows the size of the text section of all three variants versus the number of worker threads. Additionally, it shows the PAL's share by plotting the GEN variant with the text section of the PAL removed. In the case of a single worker thread, we see the overhead of TL's scheduler once again. But we also see that with 101 bytes per worker TL has the lowest slope of all. The explanation for this is that increasing the number of worker threads only involves starting another application thread, while the rest of TL's code is generic in that sense and can be reused. For NAT, we have a slope of 187 bytes per worker. This originates from the additional protothreads that run the extra workers. Although they share common code, the basic stub of each protothread is always required.

And finally for GEN, we see the biggest slope of 619 bytes per worker including the PAL and 559 bytes per worker without the PAL. The share added by the PAL has the same origin as in the NAT case. And the share added by the E-code application reflects the trade-off that we have chosen for the translation scheme: By inlining critical functions, we save CPU cycles and RAM, but we pay in the case of re-entrant critical functions. For applications like DCA and COAP, which have almost no re-entrant code, this trade-off yields good results. The thread pools of the RPC application are, however, a worst case scenario for it, because virtually all critical functions are re-entrant.

Concerning RAM consumption, however, the compiler-assisted approach still shows its strengths. In Figure 6.3c we see that GEN has almost the same slope as NAT, i.e., 182 vs. 158 bytes per worker. In contrast, TL has a slope of 310 bytes per worker which has two major reasons. First, as already explained, each thread's stack needs an extra margin to be able to host any occurring interrupt handler functions. And second, the generic nature of a thread library indeed saves in binary size, but it pays in RAM because it has to support all possible cases. For example, given a thread that only calls one particular blocking function, its meta information as stored by the scheduler still provides space for all other blocking functions because the run-time system cannot know that they will actually never be called. Likewise, a thread library cannot distinguish critical variables from non-critical ones which is why it keeps all local variables on the preallocated stacks. GEN in contrast shares RAM for non-critical variables and uses preallocated storage only for critical ones. Given the limited RAM size of many platforms, those savings are significant and the lack of static analysis is what kept thread libraries from improving their performance [89]. With compiler-assisted thread abstractions we aim to exploit exactly this possibility.

6.5. Discussion

The performance evaluation results mainly show three things. First, the thread-based programming paradigm leads to more compact code compared to event-based programming, which is an indication for what is generally perceived as being an easier programming model. Second, the performance overhead of generated E-code compared to hand-written event-based code is very minor, making the comfort of threads affordable for resource-constrained WSN devices. And third, the run-time performance of generated E-code is better than what can currently be achieved with thread libraries. All of these aspects strengthen our initial motivation for this work, and we think that the comfort of a comprehensive thread abstraction justifies the small overhead of the E-code. Also, we argue that although a dedicated threads-to-events compiler is a huge initial effort, the achieved performance makes it worthwhile.

This is particularly true when considering further optimizations that would improve the efficiency of E-code even more. For example, the Front-end could perform dead code elimination, constant folding, and other standard optimizations on the IR of each critical function. Also, E-frames can be compacted by having non-critical variables share memory if they are not both used on the same code path from one yield point to a subsequent one. Finally, E-code applications involve quite some temporary variables that could be avoided sometimes. For example, if the return value of a critical function is the result value of a critical call, then the normalization into second normal form introduces a temporary variable which is immediately copied to the result variable of the function's T-frame. A more advanced translation could avoid this temporary variable and copy the return value from the callee's T-stack to the return value of the caller's T-stack directly.

Likewise, the Back-end can perform inter-procedural optimizations. For example, if a critical function is only called once in the whole T-code application, the continuation variable can be omitted from its T-frame and its body can be inlined, avoiding the two `goto` statements to call that function and return from it. Similarly, if a function has only one callee and if one of its function parameters is only read and never modified, this parameter can be omitted from the T-frame of that function and the function can access the required value from the T-frame of its callee instead. Also, function parameters that are non-critical variables could be stored on the E-stack instead of the T-stack, which makes calling critical functions more involved, but is more memory efficient.

In general, E-code generation could be adapted such that optimizations of the employed E-code compiler are triggered. The obvious example is to preserve `switch` statements instead of substituting them with a sequence of `if` statements. More precisely, while a `switch` statement still would have to be modelled by a number of basic blocks, the IR could have additional meta information so that the Back-end is able to turn them back into a `switch` statement. Other examples include restoring `while` and `for` loops as well, using the *const type qualifier* [C99: 6.7.3] whenever appropriate, and supporting GNU C annotations for variables and type declarations.

The results of the evaluation also show that the performance of the translation scheme depends on application properties. We have seen that the RPC application is a worst case for the current translation scheme because it makes heavy use of re-entrant functions. Obviously, a different translation scheme should be used in such a case, either selected by a compiler switch or determined by an automatic classification of the input application. In [8] we have examined a different translation scheme that avoids inlining of critical functions and performs a run-time lookup to retrieve the set of local variables that belong to the current thread. Other translation schemes could experiment with dynamically allocated stack frames similar to what Capriccio does [139]. Although we do not expect dynamic memory allocation to be a good approach for embedded systems in general, there might be a class of applications for which it is a good choice nevertheless. We think that there are many more possibilities that could be explored to utilize the potential of compiler-assisted thread abstractions to its maximum. Also, a production-ready T-code compiler would have to include support for all the features that our prototype rejects to translate. This not only includes the excluded C99 features and GNU C extensions for critical functions, but also a compatibility mode for E-code compilers that do not support GNU C extensions, and support for multiple translation units.

Overall, the development and the evaluation of the translation scheme and its implementation gave us two major insights. First, the set of applied optimizations can be partitioned into two groups: optimizations that require knowledge about the thread semantics and optimizations that don't. Existing E-code compilers provide pretty good implementations for the latter group, so ideally a T-code compiler should only get involved in the former one. However, with the current architecture of chaining T-code and E-code compiler, this separation is not always possible, which is our second major insight. If the T-code compiler fails to remove unused or avoidable temporary variables, which is a standard optimization performed by E-code compilers, these variables become part of E-frames and T-frames. As their storage duration is static instead of automatic, the E-code compiler is unable to remove them.

These considerations have led to a vision of a modified compilation architecture. In this architecture, the T-code compiler still analyzes the T-code application to extract the required information. But instead of applying the threads to events transformation on it, the E-code compiler is invoked on the T-code application next. The result is an assembly of the T-code applications to which all standard optimizations have been applied. The T-code compiler then performs the transformation from threads to events on this representation, which has the additional benefit that the T-code compiler does not have to cope with the complexity of C99 and GNU C but with a rather clear execution semantic of the respective assembler language instead.

We think that the LLVM framework [72] lends itself nicely to this approach, because it provides a type safe and platform-agnostic assembler language [1], a framework to implement code optimizations, and a substantial set of existing optimizer passes. This not only permits the T-code compiler to still perform a generic transformation independent of the mote hardware, but also the generated E-code could subsequently by optimized by the same LLVM optimizers. Thus, the T-code compiler could focus on translating threads to events and on performing optimizations that require knowledge about the thread semantics. It is, however, currently unclear to us how supporting fault diagnostics of T-code applications could be achieved with this compilation architecture. This could be an interesting question for future research in this field.

6.6. Evaluating the Debugger

Evaluating a debugger is very different from evaluating a compiler. In the literature, there are mainly three different methodologies to this end: measuring interesting system properties [125, 131, 146, 147], finding a priori unknown software faults [131], and conducting a user study [146]. For the evaluation of the T-code debugger, we have, however, chosen a different methodology, as none of the usual ones is applicable in our case.

First, there are no meaningful measurements that would reflect interesting system properties. The reason for this is that the T-code debugger is designed to run on a host system, controlling an E-code debugger that takes care of the rest. Thus, things like probe effects, resource consumption of the debugger target, and communication overhead are determined by the E-code debugger only.

Second, finding unknown software faults requires existing code that uses our abstraction, which of course can not exist yet, as we are just introducing our abstraction to the community. To compensate for this, one could consider to port existing WSN applications to our abstractions. This methodology is, however, disputable, since porting can introduce new software faults or hide existing ones. This is particularly true if the initial programming abstraction differs greatly from the T-code abstraction, as protothreads do for example [31]. Apart from protothreads applications there are, however, no WSN applications that both make use of cooperative threads and are used in a deployment — at least, we are not aware of any that are publicly available. We suspect that this is due to the fact that existing thread abstractions either provide incomplete semantics or require too much resources, which is one of the initial motivations for our work.

And third, user studies, which could investigate the usability of the user interface or the effectiveness of the employed fault diagnosis technique, are out of the scope of this work. Our goal is not to advance the state-of-the-art in source-level debugging, which is why the T-code debugger resembles existing debuggers such as the GDB [129] and Eclipse [132]. These tools have been used productively and professionally for decades, which is why we do not expect any new insights from studying the effectiveness of their features. We also do not claim to provide a user-friendly application.

Instead, we only claim that completing the programming abstraction by enabling fault diagnosis for the thread-based application that runs on an event-based OS is possible. We think that the only meaningful way to support this claim is to present a proper prototype and verify its correctness, for instance via tests. This is why we provide an additional implementation of Ruab's Front-end interface. This front-end is special, as it consists of a script interpreter that allows the simulation of user interaction by running arbitrary test scripts.

Each script executes a single debugging session while focusing on a particular feature of the T-code debugger. It starts with a single command and follows with an chronologically ordered list of expected responses. Each response is possibly augmented with a single command, which is issued when the expected response is received. Additionally, an

```
 |__attribute__((tc_thread)) void collect() {
 |  uint32_t now = get_time();
 |  uint16_t value;
A|  while(true) {
 |    sleep(now + dt);
C|    now += dt;
 |    value = read_sensor();
 |    log((uint8_t*)&value, sizeof(value));
 |  }
 |}
```

```
 |__attribute__((tc_thread)) void receive() {
B|  while(true) {
 |    uint8_t buffer[10];
 |    size_t len;
 |    receive(buffer, sizeof(buffer), &len);
 |    log(buffer, len);
 |  }
 |}
```

Figure 6.4.: *Debugging with thread filter:* A, B, and C are breakpoints. Whether B is hit when resuming from A depends on the thread filter.

expected response carries values which are compared with the values of the factual response from Ruab's Core module. By these means, the test scripts can verify that the various features of the debugger work as intended — at least in the case of the employed example application, which is the DCA application (cf. Section 6.1.2).

We have written various scripts to test the different features of the T-code debugger. For example, we test breakpoints in critical functions with and without local thread filters, as well as breakpoints in auxiliary functions. These tests cover features that an E-code debugger would also provide. Tests that are more interesting verify that expressions involving local variables both for critical and auxiliary functions can be evaluated. A user of the E-code debugger would have to understand the renaming scheme that the T-code compiler applies to various identifiers to be able to issue correct queries. With the T-code debugger at hand, the user does, in contrast, not care about renaming of identifiers, as he or she can simply use the T-code identifiers for the queries.

A last group of tests cover a second major advantage of the T-code debugger, which is following the control flow at yield points. Figure 6.4 shows a simplified excerpt of the data collection application used in the evaluation, and the tests install the three breakpoints A, B, and C as indicated. Between breakpoint A and C there is a critical call to the function `sleep`. If breakpoint A is hit and the execution is continued, the behavior of the debugger depends on the status of the thread filter. If the filter is open, then breakpoint B is hit next, thus switching the context from the `collect` task over to the `receive` task. This is just what a user of an E-code debugger would experience. If the filter is limited to the `collect` task, breakpoint B is skipped instead, and breakpoint C is hit next. Thus, the context and hereby the user focus stays on the current task, enabling him or her to "step" through the `collect` function while inspecting the values of the local variables.

Overall, the tests show and verify that the T-code debugger sustains the abstraction level of the T-code during fault diagnostics.

6.7. Summary

In this chapter we described the setup of a series of experiments which we conducted with our prototypes. We also presented the experimental results.

In case of the compiler prototype, these results showed three things. First, thread-based programming leads to more compact code compared to the event-based paradigm. Second, our compiler-assisted thread abstraction is clearly more efficient that run-time based thread libraries. And third, the resource-wise costs of our compiler-assisted abstraction is very minor.

In case of the debugger, the evaluation mainly showed the correctness of both the concepts as well as the implementation of our fault diagnostics tool for T-code applications.

Overall, the contribution of this chapter was a reproducible verification and evaluation of our compiler-assisted approach to thread-based programming.

7. Conclusions

The primary goal of this thesis was to overcome the seemingly inherent trade-off between expressiveness and completeness of a thread abstraction for resource-constrained systems. We established our work in the context of wireless sensor networks (WSN). In this domain, Moore's Law is applied towards reducing costs, size, and power consumption instead of the usual increase in capabilities. It therefore constitutes an extreme environment with respect to the scarcity of resources. We additionally assume that limited resources will stay a major design requirement in this domain, particularly because the energy density and capacity of batteries is not expected to increase significantly in the future.

We have exemplified why the predominant paradigm of event-based programming makes development, testing, deployment, and operations expensive and gives rise to security issues. Since the complexity of deployed WSN applications is constantly escalating, and because they interact with the real world and become connected to the Internet, software faults are an increasingly demanding problem in this domain. Thread-based programming provides a higher abstraction level via sequential control flows and synchronous functions, and it is known to overcome most of the issues of event-based programming. The challenge is to provide a comprehensive abstraction with an efficient implementation that meets the scarce resources of WSN devices.

In this thesis we present a solution to this problem. The results of our performance evaluation yield three major results. First, they support our claim that thread-based programming provides a higher abstraction level than event-based programming. They also confirm the general opinion that thread abstractions have to trade their completeness to facilitate an efficient implementation. However, this is only true for traditional approaches that implement threads at run-time. Compiler-assisted thread abstractions can, in contrast, overcome this trade-off. This was the initial motivation of our work. The evaluation results show that, with our approach, thread-based programming can in fact be almost as efficient as event-based programming. The overhead of RAM is approximately 1 %, for ROM below 3 %, and concerning CPU cycles the overhead is below 2 %. This is the primary result of this thesis.

Compiler-assisted approaches exploit the duality of threads and events by employing a dedicated compiler that turns a thread-based application into an equivalent event-based application. This approach enables efficient implementations because the run-time system remains event-based. Additionally, compilers can analyze application properties and apply optimizations specific to the application.

We have chosen cooperative threads over preemptive ones for two reasons. First, cooperative threads minimize concurrency issues due to implicit critical sections. And second, because cooperative threads cannot be interrupted at any time, the associated state per thread can be kept comparatively small. The consequence of this choice, however, is that our work only targets the application and the service layer of WSN applications, as cooperative threads cannot provide timing guarantees.

In order to develop a translation scheme from thread-based T-code to event-based E-code, we defined the execution semantics of T-code and E-code applications. Furthermore, as the translation should turn a valid T-code application into an equivalent E-code application, we define what equivalence means on basis of the observable behavior of the two applications. The core of the translation scheme that we present in this thesis consists of two parts. The translation of the control flow turns critical functions into event handlers, while preserving the order of language statements. And the translation of the data flow replaces local variables with variables that are statically allocated, thus preserving their values over subsequent invocations of event handlers. Static analysis hereby assists in spending preallocated storage only when needed, keeping variables on the shared hardware stack whenever possible.

Ocram is our prototypical implementation of the translation scheme, and we used it to thoroughly evaluate our approach. To this end, we have chosen three archetypes of WSN applications and implemented three variants of each: one that uses native events, one that uses a thread library, and one that uses Ocram. After taking various means to verify the correctness of the translation scheme, its implementation and the implementation of the nine test applications, we performed a sequence of measurements. First, the number of source code lines were counted to provide an estimate of the expressiveness of the programming abstractions. Second, we determined the size of the machine code and the size of the initialized and uninitialized data sections of the application binaries. Third, we counted the number of CPU cycles required to execute the application logic of each variant. And finally, we measured the maximum stack consumption in each case. The last two measurements where obtained by executing the nine applications in a network simulator. Overall, we took thorough care to enable a fair comparison.

The secondary goal of this thesis was to demonstrate how our thread abstraction can be sustained for fault diagnostics. We argue that, just like programming, fault diagnostics is a question of abstraction. A comprehensive programming abstraction should therefore consider the whole development cycle. Previous programming abstractions have mostly failed to do so. This breaks the abstraction by forcing the user to cope with the event-based run-time system after all. Worse, a software developer additionally has to understand the implementation details of the abstraction itself in order to identify the software fault in the abstract program via run-time observations of the generated code. We consider this a major obstacle for the adoption of programming abstractions in practice.

The challenge in supporting fault diagnostics is to sustain the abstraction level used for programming. This thesis presents a solution to this problem. We have managed to demonstrate how to provide basic features of source-level debugging on the abstraction level of threads. The user is never confronted with the generated event-based application and can therefore reason about the application on an abstract level.

In order to preserve the abstraction level, the compiler has to gather information that can be later used by the debugger to undo the performed mapping. This approach is well-known and followed by many existing compiler tool chains. We adapted it to fit the requirements of cooperative threads. We identified necessary debugging information and investigated how to collect them during compilation.

We added these capabilities to Ocram. It particularly logs how to map source code rows and how to map variable names. Ruab is our prototypical implementation of the debugger that uses this information to perform a back-mapping from E-code to T-code at run-time. To this end, it controls an instance of an E-code debugger, which is a standard source-level debugger for C such as the GDB. Besides supporting breakpoints, evaluation of expressions and a few other basic commands, Ruab provides a thread filter. This enables software developers to follow the control flow of a single thread across yield points.

We verified the functionality of Ruab via automated tests that emulate user interaction. The tests cover all features of Ruab on the basis of one of the case study applications.

Future work in the area of this thesis could investigate improved optimizations to utilize the potential of compiler-assisted thread abstractions to its maximum. First, standard optimizations such as constant folding and dead code elimination are known to improve the efficiency of generated code. We additionally envision various optimizations specific to translating threads to events. Examples include avoiding temporary variables, compacting E-frames, and determining continuations at compile-time whenever possible. Also, the back-end could be tailored towards targeting optimizations of the E-code compiler.

Another interesting research direction would be the investigation of alternative translation schemes. The translation scheme of this thesis trades CPU cycles and RAM consumption for binary size. While the evaluation shows that this seems to be a good choice in general, it has also revealed that applications with mainly re-entrant code constitute a worst case. A future compiler should therefore be equipped with multiple translation schemes. Heuristics could determine relevant application properties in order to select the most efficient translation scheme. We expect it to be challenging to generalize the debugging information to support fault diagnostics independent of the selected translation.

When we started investigating compiler-assisted thread abstractions we did not anticipate how complex it is to provide a tool that reliably rejects invalid code. With our experience today, we think that translating threads to events on the level of (platform independent) assembler code is a promising alternative. The major advantage would be that existing compilers would deal with the complexity of C99 and GNU C. Also, existing optimization passes could be used, both before and after translating threads to events. It is, however, an open question to us how to still enable fault diagnostics on the level of the C program.

We believe that Ocram is a practical solution for three reasons. First, generated code integrates seamlessly with existing event-based code. Then, the set of supported features already allows for the development of realistic WSN applications. Nevertheless, the current implementation rejects many features that could be supported in the future. And finally, Ocram can easily be ported to other event-based kernels, because implementing a thin platform abstraction layer is a one-time effort. Ruab is also conceptually feature complete, but would require additional work to become user friendly. Overall, we think that this thesis improves the state-of-the-art in providing programming abstractions for resource-constrained systems.

A. Abstract Syntax Tree Types

This section is intended to define the terms used in Section 3.5 and to clarify the various steps of the compiler pipeline. To this end, it shows a subset of the C grammar as it is used by the parser (cf. Section 3.5.2).

The listing is an alphabetically sorted excerpt of the data type definitions of the Language.C library [57]. Therefore, is uses the Haskell syntax for algebraic data types. Please consult 2.3.2 and Chapter 4 of the Haskell Language Report 2010 [4] for clarification.

Most types are polymorphic with respect to the type of the attached annotation, i.e., the type parameter a. See Chapter 4.2.2.1 for details.

The top-level type is `CTranslationUnit`.

```
data CBuiltinThing a
  = CBuiltinVaArg              -- '__builtin_va_arg'
      (CExpression a)          --   expression
      (CDeclaration a)         --   type
      a
  | CBuiltinOffsetOf           -- '__builtin_offsetof'
      (CDeclaration a)         --   type
      [CPartDesignator a]      --   designator list
      a
  | CBuiltinTypesCompatible    -- '__builtin_types_compatible_p'
      (CDeclaration a)         --   left hand side
      (CDeclaration a) a       --   right hand side

data CCompoundBlockItem a
  = CBlockStmt                 -- statement
      (CStatement a)
  | CBlockDecl                 -- local declaration
      (CDeclaration a)
  | CNestedFunDef              -- nested function (GNU C)
      (CFunctionDef a)

data CDeclaration a
  = CDecl                      -- declarations
      [CDeclarationSpecifier a] --   specifiers (type, storage, ...)
      [(                       --   list of declarations
        Maybe (CDeclarator a)  --     declared object
      , Maybe (CInitializer a) --     initial value
      , Maybe (CExpression a)  --     size
      )]
      a

data CDeclarationSpecifier a
  = CStorageSpec (CStorageSpecifier a)
  | CTypeSpec    (CTypeSpecifier a)
  | CTypeQual    (CTypeQualifier a

data CDeclarator a
  = CDeclr                      -- declaration of an object
    (Maybe Ident)               --   name
    [CDerivedDeclarator a]      --   indirections
    (Maybe (CStringLiteral a))  --   assembler names
    [CAttribute a]
    a

data CDerivedDeclarator a
  = CPtrDeclr                   -- pointer declarator
      [CTypeQualifier a]        --   pointer type
      a
  | CArrDeclr                   -- array declarator
      [CTypeQualifier a]        --   array type
      (CArraySize a)            --   array size
      a
  | CFunDeclr                   -- function declarator
      (Either                   --   parameters
        [Ident]                 --   parameter names (K&R-style)
        (                       --   parameters (new style)
          [CDeclaration a]      --     declaration
        , Bool                  --     flag
        )
      )
      [CAttribute a] a

data CEnumeration a
  = CEnum                       -- enumeration
      (Maybe Ident)             --  name
      (Maybe [(                 --  enumeration constants
          Ident                 --    name
        , Maybe (CExpression a) --    value
        )]
      )
      [CAttribute a]
      a

data CExpression a
  = CComma                      -- comma operator
    [CExpression a]             --   expressions, n >= 2
    a
  | CAssign                     -- assignemnt operation
      CAssignOp                 --   operator
     (CExpression a)            --   lvalue
     (CExpression a)            --   rvalue
     a
  | CCond
      (CExpression a)           -- conditional
      (Maybe (CExpression a))   --   true-expression (opt. in GNU C)
      (CExpression a)           --   false-expression
                 a
    | CBinary                   -- binary operation
      CBinaryOp                 --   operator
      (CExpression a)           --   left hand side
      (CExpression a)           --   right hand side
      a
    | CCast                     -- type cast
      (CDeclaration a)          --   type name
      (CExpression a)           --   operant
      a
    | CUnary                    -- unary operation
      CUnaryOp                  --   operator
      (CExpression a)           --   operant
      a
    | CSizeofExpr               -- 'sizeof(expr)'
      (CExpression a)           --   operant
      a
    | CSizeofType               -- 'sizeof(type)'
      (CDeclaration a)          --   operant
      a
    | CAlignofExpr              -- '__alignof__(expr)' (GNU C)
      (CExpression a)           --   operant
      a
    | CAlignofType              -- '__alignof__(type)' (GNU C)
      (CDeclaration a)          --   operant
      a
    | CComplexReal              -- real part of complex number
      (CExpression a)           --   operant
      a
    | CComplexImag              -- imaginary part of complex number
      (CExpression a)           --   operant
      a
    | CIndex                    -- array subscripting
      (CExpression a)           --   array
      (CExpression a)           --   index
      a
    | CCall                     -- function call
      (CExpression a)           --   callee
      [CExpression a]           --   parameters
      a
    | CMember                   -- member access
      (CExpression a)           --   structure
      Ident                     --   member name
      Bool                      --   de-reference? (true => '->')
      a
    | CVar                      -- variable or enumeration constant
      Ident                     --   name
      a
    | CConst                    -- literal
      (CConstant a)
```

```
  | CCompoundLit                 -- compound literal
      (CDeclaration a)           --   target type
      (CInitializerList a)       --   initialiser list
      a
  | CStatExpr                    -- statement as expression (GNU C)
      (CStatement a) a
  | CLabAddrExpr                 -- address of a label (GNU C)
      Ident                      --   label name
      a
  | CBuiltinExpr                 -- built-in expressions (GNU C)
      (CBuiltinThing a)

data CExternalDeclaration a
  = CDeclExt                     -- global declaration
      (CDeclaration a)
  | CFDefExt                     -- function definition
      (CFunctionDef a)
  | CAsmExt                      -- assembler code
      (CStringLiteral a) a

data CFunctionDef a
  = CFunDef                      -- function definition
      [CDeclarationSpecifier a] --   return type and specifiers
      (CDeclarator a)            --   the declarator
      [CDeclaration a]           --   parameters (K&R-style only)
      (CStatement a)             --   body
      a

data CInitializer a
  = CInitExpr                    -- assignment expressoin
      (CExpression a) a
  | CInitList                    -- initilizer list
    (CInitializerList a) a

type CInitializerList a = [(    -- initializer list
    [CPartDesignator a]          --   desigators
  , CInitializer a               --   value
  )]

data CPartDesignator a
  = CArrDesig                    -- array position designator
      (CExpression a) a          --   index
  | CMemberDesig                 -- member designator
      Ident a                    --   name of the member
  | CRangeDesig                  -- array range designator (GNU C)
      (CExpression a)            --   from
      (CExpression a)            --   to
       a

data CStatement a
  = CLabel                       -- label
```

```
      Ident                    --   name
      (CStatement a)           --   subsequent statement
      [CAttribute a]
      a
  | CCase                      -- case statement
      (CExpression a)          --   constant expression
      (CStatement a)           --   subsequent statement
      a
  | CCases                     -- case range (GNU C)
      (CExpression a)          --   lower bound
      (CExpression a)          --   upper bound
      (CStatement a)           --   subsequent statement
      a
  | CDefault                   -- default statement
      (CStatement a) a
  | CExpr                      -- expression statement
      (Maybe (CExpression a))
      a
  | CCompound                  -- block/scope
      [Ident]                  --   not used
      [CCompoundBlockItem a]   --   contained block items
      a
  | CIf                        -- if statement
      (CExpression a)          --   condition
      (CStatement a)           --   'then' branch
      (Maybe (CStatement a))   --   'else' branch
      a
  | CSwitch                    -- switch statement
      (CExpression a)          --   controlling expression
      (CStatement a)           --   body
      a
  | CWhile                     -- do or while loop
      (CExpression a)          --   condition
      (CStatement a)           --   body
      Bool                     --   true => do loop
      a
  | CFor                       -- for loop
      (Either                  --   initializer
        (Maybe (CExpression a))
        (CDeclaration a)
      )
      (Maybe (CExpression a))  --   controlling expression
      (Maybe (CExpression a))  --   increment expression
      (CStatement a)           --   body
      a
  | CGoto                      -- goto
      Ident                    --  target
      a
  | CGotoPtr                   -- computed goto (GNU C)
      (CExpression a)          -- target
      a
  | CCont a                     -- 'continue'
  | CBreak a                    -- 'break'
  | CReturn                     -- return
      (Maybe (CExpression a))   --  return value
      a
  | CAsm                        -- assembly statement
      (CAssemblyStatement a) a

data CStorageSpecifier a
  = CAuto     a                 -- 'auto'
  | CRegister a                 -- 'register'
  | CStatic   a                 -- 'static'
  | CExtern   a                 -- 'extern'
  | CTypedef  a                 -- 'typedef'
  | CThread   a                 -- '__thread': GNU C thread local storage

data CStructTag
  = CStructTag                  -- 'struct'
  | CUnionTag                   -- 'union'

data CStructureUnion a
  = CStruct                     -- structure or union
      CStructTag                --  'struct' or 'union'
      (Maybe Ident)             --  name
      (Maybe [CDeclaration a])  --  member declarations
      [CAttribute a]
      a

data CTranslationUnit a
  = CTranslUnit                 -- translation unit
      [CExternalDeclaration a]  -- external (global) declarations
      a

data CTypeQualifier a
  = CConstQual a                -- 'const'
  | CVolatQual a                -- 'volatile'
  | CRestrQual a                -- 'restricted'
  | CInlineQual a               -- 'inline'
  | CAttrQual                   -- '__attribute__((name))'
      (CAttribute a)

data CTypeSpecifier a
  = CVoidType    a              -- 'void'
  | CCharType    a              -- 'char'
  | CShortType   a              -- 'shor t'
  | CIntType     a              -- 'int'
  | CLongType    a              -- 'long'
  | CFloatType   a              -- 'float'
  | CDoubleType  a              -- 'double'
  | CSignedType  a              -- 'signed'
  | CUnsigType   a              -- 'unsigned'
| CBoolType    a              -- '_Bool'
| CComplexType a              -- '_Complex'
| CSUType                     -- struct or union
    (CStructureUnion a)
    a
| CEnumType                   -- Enumeration specifier
    (CEnumeration a)
    a
| CTypeDef                    -- typedef name
    Ident                     --   type name
    a
| CTypeOfExpr                 -- 'typeof(expr)'
    (CExpression a)
    a
| CTypeOfType                 -- 'typeof(type)'
    (CDeclaration a)
    a
```

B. Short-circuit Evaluation

The `short_circuit` function from Section 3.5.4.3 is based on two sub-functions, namely `traverse` and `transform`.

Function B.1: `traverse ::`
```
    CExpression      --  an arbitrary expression
 →  (                --  a tuple consisting of
      CExpression,       --  a substituting expression, and
      [CStatement],      --  a list of statements emulating
                             short-circuit behavior (if
                             required), and
      [Variable]         --  a list of new variables
    )
```

Function B.2: `transform ::`
```
    CStatement    --  a basic statement
 →  [CStatement]  --  a list of statements emulating
                      short-circuit behavior (if required)
```

`traverse` takes an expression and returns a new expression, a list of statements and a list of IR variables. The new expression can replace the old one if the given statements are executed in advance. Also, the new expression makes use of the returned new variables. `traverse` is a recursive function, as expressions are recursive themselves. If there is a base case such as a variable or a constant at hand, the new expression equals the old one and the list of statements and the list of variables is empty. In most other cases, the expression is decomposed into its subexpressions, `traverse` is applied to these subexpressions in turn, and the recursive results are composed into a new expression and concatenated into the list of statements and list of variables. The interesting case is the one of a binary operation that uses a logical operator and contains a critical call (cf. Figure 3.13).

In such a case, `traverse` is again applied recursively to the subexpressions, but the results are composed differently. First of all, the new expression is just a new variable of type `_Bool`, whose IR is returned as the single element of the list of new variables. Additionally, the list of returned statements is built up as follows: It starts with the list of traversed statements of the left-hand side. According to the semantics of `traverse`, these statements have to be executed before the new expression of the left-hand side may be used. This expression is assigned to a new Boolean variable next. A subsequent `if` statement redirects the control flow to the end of the list of statements that is currently built if the value of the Boolean variable is `false` in case of a logical and-expression or `true` in case of a logical or-expression. Otherwise, the control flow continues with the next statement, which is the start of the list of traversed statements of the right-hand side. Again, these statements have to be executed before the new expression of the right-hand side may be used. This expression is finally assigned to the Boolean variable. Overall, if this list of statements is executed, the Boolean variable, which resembles the returned new expression, holds the value of the logical operation. Thus, the semantics of `traverse` are implemented correctly. At the same time, the right-hand side of the logical operation is not evaluated when indicated, which is how the short-circuit semantics are implemented.

The `transform` function matches all possible statements, calls the function `traverse` on the interior expressions, and composes from the results a new statement, which is prepended by the list of statements returned by `traverse`. Note that if none of the sub-expressions of a statement contains a critical call, `transform` effectively returns the same statement and the list of new variables is empty. In Figure 3.13a, however, line 1–4 are the list of statements returned by `traverse` when invoked on "`c() || y`" and line 5 is the original statement with the logical operation replaced by the new expression that was returned by `traverse`. In Figure 3.13b, line 1–8 resemble the list of statements returned by `traverse` when invoked on the whole expression of the `return` statement. Also, line 9 shows that return statement with the new expression substituting the old one. Recursively, line 1–4 are the list of statements returned by `traverse` when invoked on the left-hand side of the logical or-expression. And line 5 shows how the new expression returned by the same `traverse` invocation is used to the new Boolean variable, just like line 1 one level down the recursion. Overall, `short_circuit` invokes `transform` on each input statement while collecting the new variables.

C. Reproducing the Experiments

This section is intended to enable independent third parties to reproduce and verify our experimental results. We specify the version of each involved software component, and we document each step we took to produce the results. Different version of the employed software components or alternative implementations might work as well, so the documented dependencies are not strict.

The following sections assume to be processed in order. Each section will start with a list of dependencies followed by a sequence of shell commands. We used the Z shell (`http://zsh.sourceforge.net/`) on Ubuntu GNU/Linux (maverick, 64-bit) (`http://www.ubuntu.com`) to execute these commands.

Some files contain hard-coded path names which reflect our local installation. In order to reproduce the experiments in different environments these paths have to be adapted. A full list is provided at the end of this section.

Ocram

This section documents how to build the Ocram binary.

Dependencies

- git 1.7.1 - version control
  `http://git-scm.com`
- Haskell Platform 2012.2.0.0 - basic development environment
  `http://www.haskell.org/platform`
- GHC 7.4.1 - Haskell implementation
  `http://www.haskell.org/ghc`
- cabal-dev 0.9.1 - sandbox management
  `http://github.com/creswick/cabal-dev`

Steps

```
# download the Ocram project
$ git clone git://github.com/copton/ocram.git
# enter the Ocram project tree
$ cd ocram
# remember project path
$ export OCRAM=`pwd`
# switch to the thesis branch
$ git checkout thesis
```

```
# enter build environment
$ source ./setup
# enter Ocram source tree
$ cd ocram
# build the binary
$ cabal-dev install
# execute the test suite
$ ./cabal-dev/bin/ocram -test
```

Contiki

This section documents how to prepare Contiki.

Dependencies

- Contiki 2.5.x - "The Open Source OS for the Internet of Things"
  `http://contiki-os.org`
- JDK 1.6.0_30 - Java Development Kit
  `http://www.oracle.com/technetwork/java/javase`
- Ant 1.7.1 - build system
  `http://ant.apache.org/`

Steps

```
# download the Contiki project
$ git clone git://contiki.git.sourceforge.net/gitroot/contiki/contiki
# enter Contiki project tree
$ cd contiki
# remember project path
$ export CONTIKI=`pwd`
# checkout the correct commit
$ git checkout -b local 92765b384ec87de96e97529dd463c48a8d199ec3
# apply patches
$ git apply $OCRAM/patches/contiki/*.patch
# enter Cooja path
$ cd tools/cooja
# build Cooja
$ ant build
```

Our patches to Contiki are:

1. `local-modifications.patch`: disable radio duty cycling and add our Cooja plugin to both Cooja and MSPsim configuration
2. `OcramCoojaPlugin.patch`: add a symbolic link of the Ocram plugin to the `mspsim` directory.

3. `disable-wrong-assertion.patch`: disable an assertion that is spuriously triggered in head-less executions of Cooja.

4. `detailed-CPU-cycle-tracking-for-processes.patch`: introduce the `process_hook` API of our Ocram plugin and instrument the MSP430 platform to enable accurate measurement of CPU cycles.

Measurements

This section documents how to run the experiments and obtain the results.

Dependencies

- gcc-msp430 3.2.3 - MSP430-port of the GCC
  `http://mspgcc.sourceforge.net`

- binutils-msp430 2.17-2 - MSP430-port of GNU binutils
  `http://mspgcc.sourceforge.net`

- GNU Make 3.81 - build system
  `http://www.gnu.org/software/make/`

- schroot 1.4.7 - chroot management
  `http://wiki.debian.org/Schroot`

- Python 2.6.6 - dynamic scripting language
  `http://www.python.org`

- Jinja2 2.5.2 - template language
  `http://jinja.pocoo.org/docs/`

- GNU Plot 4.4 - graphing utility
  `http://www.gnuplot.info/`

Steps

The build system assumes a `schroot` environment called `contiki` which includes the installation of the 32-bit MSP430 tool chain.

```
# enter Ocram project
$ cd $OCRAM
# enter build environment
$ source ./setup
# enter case study application path
$ cd applications/contiki
# build applications
$ make
# run experiments
$ make plot
```

If everything goes right, the results can be found in
`$OCRAM/applications/contiki/{dca,coap,rpc?}/bench.results`
and the plots can be found in
`$OCRAM/applications/contiki/plots/*.png`.

Ruab

This sections document how to build and test Ruab.

Dependencies

- GDB 7.4-2012.02 - The GNU Debugger
  `http://http://www.gnu.org/software/gdb/`

Steps

The tests assume a `schroot` environment called `precise` which includes the installation of the GDB.

```
# enter Ocram project
$ cd $OCRAM
# enter build environment
$ source ./setup
# enter simulation OS path
$ cd applications/simulation_os
# enter build environment
$ source ./setup-linux
# build OS and case study applications
$ make
# test OS and case study applications
$ make test
# enter Ruab tree
$ cd $OCRAM/ruab
# build binary
$ cabal-dev install
# run tests
$ ./cabal-dev/bin/ruab -test
```

Hard-coded Paths

The following files contain hard-coded paths that have to be adapted for different environments:

- `$CONTIKI/tools/cooja/apps/mspsim/src/mspmote/`
  `plugins/OcramCoojaPlugin.java`
  is a symbolic link to
  `$OCRAM/applications/contiki/OcramCoojaPlugin.java`.

- `$OCRAM/setup` contains references to the Haskell installation.
- `$OCRAM/applications/contiki/`
  `{coap,dca,rpc2}/{generated,native,runtime}`
  `Makefile{,.contiki}`
  contain references to `$CONTIKI`.
- `$OCRAM/ruab/src/Ruab/Backend/GDB/Test.hs` contains references to `$OCRAM`.
- `$OCRAM/applications/contiki/test.py` contains a reference to `$CONTIKI`.

D. Free and Open Source Software

This work has been established mainly with the help of free and open source software. The spirit of free and open source software is close to the spirit of science, i.e., building upon the work of others and contributing back to the public. One major incentive for participation is reputation. Therefore, we want to gratefully mention each project that directly or indirectly enabled us in creating this work, even if the individual software licences may not force us to do so.

- Development Tools
  - **Ant**
    a build system
    `http://ant.apache.org/`
  - **cabal-dev**
    a tool for managing development builds of Haskell projects
    `http://hackage.haskell.org/package/cabal-dev`
  - **Dia**
    a diagram creation program
    `https://live.gnome.org/Dia`
  - **GDB**
    the GNU project debugger
    `http://www.gnu.org/software/gdb/`
  - **git**
    a distributed version control system
    `http://git-scm.com/`
  - **Glade**
    a RAD for GTK+ user interfaces
    `http://glade.gnome.org/`
  - **GNU Make**
    a build system
    `http://www.gnu.org/software/make/`
  - **hlint**
    lint-like behavior for Haskell
    `http://community.haskell.org/~ndm/hlint/`
  - **Vim**
    a text editor
    `http://www.vim.org`
  - **wdiff**
    a word per word difference tool
    `http://www.gnu.org/software/wdiff/`

- Text Generation Tools
  - **TEX**
    a typesetting system
    Donald E. Knuth: Digital Typography. University of Chicago Press, 1999, ISBN 1-57586-010-4
  - **Gnuplot**
    a graphing utility
    `http://www.gnuplot.info/`
  - **Inkscape**
    a vector graphics editor
    `http://inkscape.org/`
  - **LATEX**
    a document markup language and document preparation system for TEX
    `http://www.latex-project.org/`
  - **pdftex**
    a TEXdistribution
    `http://www.tug.org/applications/pdftex/`
  - **xpdf**
    a PDF suite
    `http://www.foolabs.com/xpdf/`
  - **evince**
    a document viewer
    `http://www.gnome.org/projects/evince/`
- Programming Language Implementations
  - **Glasgow Haskell Compiler**
    a compiler and interactive environment for Haskell
    `http://www.haskell.org/ghc/`
  - **GNU Compiler Collection**
    a compiler and a standard library implementation for C
    `http://gcc.gnu.org/`
  - **Jinja2**
    a full featured template engine for Python
    `http://jinja.pocoo.org/`
  - **MSP430 GCC**
    a GCC port for the MSP430 platform
    `http://mspgcc.sourceforge.net/`
  - **OpenJDK**
    an implementation of the Java Platform, Standard Edition
    `http://openjdk.java.net/`
  - **Python**
    a general-purpose, interpreted, and dynamically typed programming language
    `http://www.python.org`

  - **zsh**
    an interactive shell and scripting language
    `http://www.zsh.org/`

- Evaluation
  - **Contiki**
    "the operating system for the Internet of Things"
    `http://www.contiki-os.org/`
  - **Cooja**
    a cross-level sensor network simulator
    `http://www.contiki-os.org/`
  - **GNU binutils**
    a collection of binary tools
    `http://www.gnu.org/software/binutils/`
  - **Linux**
    Unix-like operating system kernel
    `http://www.kernel.org`
  - **schroot**
    a management tool for chroot environments
    `http://www.debian-administration.org/articles/566`
  - **TinyOS**
    an operating system for low-power wireless devices
    `http://www.tinyos.net/`

- Haskell Libraries
  - **bytestring**
    efficient, compact and immutable byte strings
    `http://hackage.haskell.org/package/bytestring`
  - **Data.Generics**
    generic programming in Haskell
    `http://hackage.haskell.org/package/syb`
  - **fgl**
    a functional graph library
    `http://hackage.haskell.org/package/fgl`
  - **gtk2hs**
    a GUI library based on GtK+
    `http://projects.haskell.org/gtk2hs/`
  - **hgdbi**
    an implementation of the GDB Machine Interface
    `http://hackage.haskell.org/package/hgdbmi`
  - **hoopl**
    higher-order optimization
    `http://hackage.haskell.org/package/hoopl`

- **HUnit**
  a unit test library
  `http://hackage.haskell.org/package/HUnit`
- **json**
  a JSON parser and printer library
  `http://hackage.haskell.org/package/json`
- **Language.C**
  analysis and generation of C code
  `http://hackage.haskell.org/package/language-c`
- **nano-md5**
  efficient bytestring bindings to OpenSSL
  `http://hackage.haskell.org/package/nano-md5`
- **parsec**
  an industrial-strength parser library
  `http://hackage.haskell.org/package/parsec`
- **pretty**
  print out text in a consistent format of your choosing
  `http://hackage.haskell.org/package/pretty`
- **stm**
  software transactional memory
  `http://hackage.haskell.org/package/stm`
- **test-framework**
  a generic test framework
  `http://hackage.haskell.org/package/test-framework`

In the context of this work we contributed to the following projects: Cooja, pretty, Language.C, hgdbmi, hlint, MSPsim

Bibliography

[1] V. Adve, C. Lattern, M. Brukman, A. Shukla, and B. R. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the Symposium on Microarchitecture*, MICRO 36, pages 205–216, 2003.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track of the USENIX Technical Conference*, pages 289–302, 2002.

[3] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A Line in the Sand: A Wireless Sensor Network for Target Detection, Classification, and Tracking. *Computer Networks*, 46(5):605–634, 2004.

[4] Arvind, L. Augustsson, D. Barton, B. Boutel, W. Burton, M. M. T. Chakravarty, D. Coutts, J. Fairbairn, J. Fasel, J. Goerzen, A. Gordon, M. Guzman, K. Hammond, B. Heeren, R. Hinze, P. Hudak, J. Hughes, T. Johnsson, I. Jones, M. Jones, D. Kieburtz, J. Launchbury, A. Löh, I. Lynagh, S. Marlow, J. Meacham, E. Meijer, R. Nanavati, R. Nikhil, H. Nilsson, R. Paterson, J. Peterson, S. Peyton Jones, M. Reeve, A. Reid, C. Runciman, D. Stewart, M. Sulzmann, A. Tang, S. Thompson, P. Wadler, M. Wallace, S. Weirich, D. Wise, and J. Young. Haskell 2010 - Language Report. `http://www.haskell.org/onlinereport/haskell2010`, 2010.

[5] Arvind, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fairbairn, J. Fasel, A. Gordon, M. Guzman, K. Hammond, R. Hinze, P. Hudak, J. Hughes, T. Johnsson, M. Jones, D. Kieburtz, J. Launchbury, E. Meijer, R. Nikhil, J. Peterson, S. Peyton Jones, M. Reeve, A. Reid, C. Runciman, P. Wadler, D. Wise, and J. Young. Haskell 98 Language and Libraries - The Revised Report. `http://www.haskell.org/onlinereport`, 1999.

[6] A. Bernauer. Ocram, A Compiler-Assisted Thread Abstraction for Resource-Constrained Systems. `http://www.github.com/copton/ocram/tree/thesis`, 2012.

[7] A. Bernauer and K. Römer. Meta-Debugging Pervasive Computers. In *Proceedings of the Workshop on Programming Methods for Mobile and Pervasive Systems*, PMMPS, 2010. 4 pp.

[8] A. Bernauer, K. Römer, S. Santini, and J. Ma. Threads2Events: An Automatic Code Generation Approach. In *Proceedings of the Workshop on Hot Topics in Embedded Networked Sensors*, HotEmNets, 2010. Article No. 8, 5 pp.

[9] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable Topology Control for Deployment-Support Networks. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 359–363, 2005.

[10] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *Mobile Networks and Applications*, 10(4):563–579, 2005.

[11] D. J. A. Bijwaard, W. A. P. van Kleunen, P. J. M. Havinga, L. Kleiboer, and M. J. J. Bijl. Industry: Using Dynamic WSNs in Smart Logistics for Fruits and Pharmacy. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 218–231, 2011.

[12] C. Bormann and Z. Shelby. Blockwise Transfers in CoAP, 2011. draft-ietf-core-block.

[13] S. Bradner. Key Words for Use in RFCs to Indicate Requirement Levels. RFC 2119.

[14] N. Burri, R. Flury, S. Nellen, B. Sigg, P. Sommer, and R. Wattenhofer. YETI: An Eclipse Plug-in for TinyOS 2.1. In *Adjunct Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 295–296, 2009.

[15] Q. Cao, T. Abdelzaher, Stankovic J., and T. He. The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 233–244, 2008.

[16] P. Castella, F. Petit, and F. Belarbi. Sugar on the Pill: Are You Suffering from the Headache of Truck Parking Management? *Thinking Highways*, 7(1):64–66, 2012.

[17] M. Ceriotti, M. Corra, L. D'Orazio, R. Doriguzzi, D. Facchin, S. Guna, G. P. Jesi, R. Lo Cigno, L. Mottola, A. L. Murphy, M. Pescalli, G. P. Picco, D. Pregnolato, and C. Torghele. Is there Light at the Ends of the Tunnel? Wireless Sensor Networks for Adaptive Lighting in Road Tunnels. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 187–198, 2011.

[18] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 148–157, 2007.

[19] B. Chen, G. Peterson, G. Mainland, and M. Welsh. Livenet: Using Passive Monitoring to Reconstruct Sensor Network Dynamics. In *Proceedings of the Conference on Distributed Computing in Sensor Systems*, DCOSS, pages 79–98, 2008.

[20] W. D. Clinger. Foundations of Actor Semantics. Technical Report AITR-633, Massachusetts Institute of Technology, 1981.

[21] M. E. Conway. Design of a Seperable Transition-Diagram Compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[22] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Programming Wireless Sensor Networks with the TeenyLime Middleware. In *Proceedings of the Middleware Conference*, pages 429–449, 2007.

[23] Crossbow Technology Inc. 41 Daggett Dr, San Jose, CA 95134. Wireless Measurement System, MICA2.

[24] R. Cunningham and E. Kohler. Making Events Less Slippery With eel. In *Proceedings of Conference on Hot Topics in Operating Systems*, HotOS, 2005. Article No. 3, 6 pp.

[25] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985. CST-33-85.

[26] H. Dubois-Ferrier, R. Meier, L. Fabre, and P. Metrailler. TinyNode: A Comprehensive Platform for Wireless Sensor Network Applications. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 358–365, 2006.

[27] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan. Adding Preemption to TinyOS. In *Proceedings of the Workshop on Embedded Networked Sensors*, EmNets, pages 88–92, 2007.

[28] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan. Improving the Energy Efficiency of the MANTIS Kernel. In *Proceedings of the European Conference on Wireless Sensor Networks*, EWSN, pages 261–276, 2007.

[29] A. Dunkels. Full TCP/IP for 8-bit Architectures. In *Proceedings of the Conference on Mobile Systems, Applications, and Services*, MobiSys, pages 85–98, 2003.

[30] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the Conference on Local Computer Networks*, LCN, pages 455–462, 2004.

[31] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 29–42, 2006.

[32] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making Sensor Networks IPv6 Ready. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 421–422, 2008.

[33] R. K. Dybvig. *The Scheme Programming Language*. Prentice Hall, 2003.

[34] V. Dyo, S. A. Ellwood, D. W. Macdonald, A. Markham, C. Mascolo, B. Pásztor, S. Scellato, N. Trigoni, R. Wohlers, and K. Yousef. Evolution and Sustainability of a Wildlife Monitoring Sensor Network. In *Proceedings of Conference on Embedded Networked Sensor Systems*, SenSys, pages 127–140, 2010.

[35] M. Eager and the DWARF Debugging Information Format Committee. DWARF Debugging Information Format - Version 4. `http://www.dwarfstd.org/doc/DWARF4.pdf`.

[36] R. S. Engelschall. Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation. In *Proceedings of the USENIX Technical Conference*, ATEC, pages 20–20, 2000.

[37] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, T. Voigt, and N. Tsiftes. MSPsim: An Extensible Simulator for MSP430-Equipped Sensor Boards. In *Adjunct Proceedings of the European Conference on Wireless Sensor Networks*, EWSN, 2008. 2 pp.

[38] J. Eriksson, F. Österlind, N. Finne, A. Dunkels, N. Tsiftes, and T. Voigt. Accurate Network-Scale Power Profiling for Sensor Network Simulators. In *Proceedings of the Conference on Wireless Sensor Networks*, EWSN, pages 312–326, 2009.

[39] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. In *Proceedings of the Real-Time Systems Symposium*, RTSS, pages 256–265, 2005.

[40] B. Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL, pages 111–122, 2004.

[41] Free Software Foundation. GNU General Public License, Version 2. `http://www.gnu.org/licenses/old-licenses/gpl-2.0.txt`, 1991.

[42] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. E. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 1–11, 2003.

[43] G. R. Gircys. *Understanding and Using COFF*. O'Reilly Media, 1988.

[44] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 1–14, 2009.

[45] J. Gosling, B. Joy, G. Steele, Bracha G., and A. Buckley. The Java^{TM} Language Specification: Java SE 7 Edition. `http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`.

[46] R. Grimm. Better Extensibility through Modular Syntax. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 38–51, 2006.

[47] Object Management Group. The Common Object Request Broker: Architecture and Specification, 1998. formal/2012-11-16.

[48] R. Gummadi, O. Gnawali, and R. Govindan. Macro-Programming Wireless Sensor Networks Using Kairos. In *Proceedings of the Conference on Distributed Computing in Sensor Systems*, DCOSS, pages 126–140, 2005.

[49] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *Proceedings of the Conference on Mobile Systems, Applications, and Services*, MobiSys, pages 163–176, 2005.

[50] Robert Hanmer. *Patterns for Fault Tolerant Software*. Wiley, 2007.

[51] T. Harris, M. Abadi, R. Isaacs, and R. MrIlroy. AC: Composable Asynchronous IO for Native Languages. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 903–920, 2011.

[52] A. Hasler, I. Talzi, J. Beutel, C. Tschudin, and S. Gruber. Wireless Sensor Networks in Permafrost Research: Concept, Requirements, Implementation, and Challenges. In *Proceedings of the Conference on Permafrost*, pages 669–674, 2008.

[53] C. T. Haynes, D. P. Friedman, and W. Wand. Obtaining Coroutines with Continuations. *Journal of Computer Languages*, 11(3/4):143–153, 1986.

[54] J. Hill and D. E. Culler. A Wireless Embedded Sensor Architecture for System-Level Optimization. Technical report, UC Berkeley, 2002.

[55] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. *ACM SIGARCH Computer Architecture News*, 28(5):93–104, 2000.

[56] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[57] B. Huber, Chakravarty M. M. T., D. Coutts, and B. Felgenhauer. Language.C - Analysis and Generation of C Code. `http://hackage.haskell.org/package/language-c`.

[58] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the Conference on History of Programming Languages*, HOPL-III, pages 12–1–12–55, 2007.

[59] J. Hughes. Why Functional Programming Matters. In D. Turner, editor, *Research Topics in Functional Programming*, pages 98–107. Addison-Wesley, 1990.

[60] J. Hughes. The Design of a Pretty-Printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 53–96. Springer, 1995.

[61] J. W. Hui and D. E. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 15–28, 2008.

[62] A. Hunt, D. Thomas, and W. Cunningham. *The Pragmatic Programmer*. Addison-Wesley Longman, 1999.

[63] International Organization for Standardization, International Electrotechnical Commission. Information Technology — Syntactic Meta Language — Extended BNF, 1996. ISO/IEC 14977:1996.

[64] International Organization for Standardization, International Electrotechnical Commission. Programming languages — C, 1999. ISO/IEC 9899:1999.

[65] International Organization for Standardization, International Electrotechnical Commission. Information Technology —– Programming Languages, their Environments and System Software Interfaces — ECMAScript Language Specification, 2011. ISO/IEC 16262:2011.

[66] B. W. Kernighan and D. M Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[67] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 254–263, 2007.

[68] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-E, P. Levis, A. Terzis, and R. Govindan. TOSThreads: Thread-safe and Non-invasive Preemption in TinyOS. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 127–140, 2009.

[69] J.G. Ko, K. Klues, C. Richter, W. Hofer, B. Kusy, M. Brünig, T. Schmid, Q. Wang, P. Dutta, and A. Terzis. Low Power or High Performance? A Tradeoff Whose Time Has Come (and Nearly Gone). In *Proceedings of the European Conference on Wireless Sensor Networks*, EWSN, pages 98–114, 2012.

[70] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proceedings of the Conference on Mobile Ad-hoc and Sensor Systems*, MASS, pages 855–860, 2011.

[71] R. Lämmel and S. Peyton Jones. Scrap Your Boilerplate: A Practical Approach to Generic Programming. In *Proceedings of the Workshop on Types in Language Design and Implementation*, TLDI, pages 26–37, 2003.

[72] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transfomation. In *Proceedings of the Symposium on Code Generation and Optimization*, GCO, pages 75–86, 2004.

[73] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *Operating Systems Review*, 13(2):3–19, 1979.

[74] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, 2006.

[75] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. *SIGPLAN Notices*, 31(1):270–282, 2002.

[76] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 126–137, 2003.

[77] P. Levis, N. Patel, D. E. Culler, and S. Shenker. Trickle: A Self-regulating Algorithm for Code Maintenance and Propagation in Wireless Sensor Networks. In *Proceedings of the Symposium on Network Systems Design and Implementation*, NSDI, pages 2–2, 2004.

[78] Philip Levis and David Culler. Maté: A Tiny Virtual Machine for Sensor Networks. *SIGOPS Operating Systems Review*, 36(5):85–95, 2002.

[79] C. Liechti, D. Diky, and P. A. Bigot. GCC Toolchain for MSP430. `http://mspgcc.sourceforge.net/`.

[80] M. Lipovača. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2011.

[81] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of the Conference on Management of Data*, SIGMOD, pages 491–502, 2003.

[82] G. Mainland, G. Morrisett, M. Welsh, and R. Newton. Sensor Network Programming with Flask. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 385–386, 2007.

[83] A. Mainwaring, D. E. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the Workshop on Wireless Sensor Networks and Applications*, WSNA, pages 88–97, 2002.

[84] C. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*. Lecture Notes in Computer Science 95. Springer, 1980.

[85] S. Marlow and S. Peyton Jones. The Glasgow Haskell Compiler. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications*, volume 2, pages 67–88. lulu.com, 2008.

[86] F. Mattern and C. Flörkemeier. From the Internet of Computers to the Internet of Things. In K. Sachs, I. Petrov, and P. Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *LNCS*, pages 242–259. Springer, 2010.

[87] T. D. May, S. H. Dunning, and G. A. Dowding. An RPC Design for Wireless Sensor Networks. *Pervasive Computing and Communications*, 2(4):384–397, 2007.

[88] W. P. McCartney and N. Sridhar. Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 167–180, 2006.

[89] W. P. McCartney and N. Sridhar. Stackless Preemptive Multi-Threading for TinyOS. In *Proceedings of the Conference on Distributed Computing in Sensor Systems*, DCOSS, pages 1–8, 2011.

[90] J. Menapace, J. Kingdon, D. MacKenzie, and Cygnus Support. The "stabs" Debug Format. `http://sourceware.org/gdb/current/onlinedocs/stabs`.

[91] W. M. Meriall, F. Newberg, K. Sohrabi, W. Kaiser, and G. Pottie. Collaborative Networking Requirements for Unattended Ground Sensor Systems. In *Proceedings of the Aerospace Conference*, pages 2153–2165, 2003.

[92] R. Milner. A Theory of Type Polymorphism in Programming. *Computer and System Science*, 17:348–374, 1978.

[93] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks, 2007. RFC 4944.

[94] J. Moses. The Function of FUNCTION in LISP or Why the FUNARG Problem Should be Called the Environment Problem. *SIGSAM Bulletin*, (15):13–27, 1970.

[95] L. Mottola, G. P. Picco, P. Valleri, F. J. Oppermann, and K. Römer. The makeSense Programming Model. Technical Report D-3.1, Swedish Institute of Computer Science, Università degli Studi di Trento, Universität zu Lübeck, 2011.

[96] L. Mottola and G. P. Pietro. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *ACM Computing Surveys*, 43(3):19:1–19:51, 2011.

[97] P. Naik and K. M. Sivalingam. A Survey of MAC Protocols for Sensor Networks. In C.S. Raghavendra, K. M. Sivalingam, and T. Znati, editors, *Wireless Sensor Networks*, pages 93–107. Kluwer Academic Publishers Norwell, 2004.

[98] G. C. Necula, S. McPeak, S. P. Rahul, and W. Westley. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[99] C. Nitta, R. Pandey, and Y. Ramin. Y-threads: Supporting Concurrency in Wireless Sensor Networks. In *Proceedings of the Conference on Distributed Computing in Sensor Systems*, DCOSS, pages 169–184, 2006.

[100] M. Odersky, P. Altherr, V. Cremet, I. Dragos, F. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[101] B. O'Sullivan, D. B. Steward, and J. Görzen. *Real World Haskell*. O'Reilly, 2009.

[102] J. K. Ousterhout. Why Threads are a Bad Idea (for most purposes), 1996. Presentation given at the Usenix Annual Technical Conference.

[103] S. Peyton Jones. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In T. Hoare, M. Broy, and R. Steinbruggen, editors, *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2000.

[104] J. Polastre, J. Hill, and D. E. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *Proceedings of the Conference on Embedded Networked Systems*, SenSys, pages 95–107, 2004.

[105] J. Polastre, R. Szewczyk, and D. E. Culler. Telos: Enabling Ultra-Low Power Wireless Research. In *Proceedings of the Symposium on Information Processing in Sensor Networks*, IPSN, pages 364–369, 2005.

[106] V. Raghunathan, A. Kansal, J. Hsu, J. Friedman, and M. Srivastava. Design Considerations for Solar Energy Harvesting Wireless Embedded Systems. In *Proceedings of the Symposium on Information Processing in Sensor Networks*, IPSN, pages 457–462, 2005.

[107] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the Sensor Network Debugger. In *Proceedings of the Conference on Embedded Networked Systems*, SenSys, pages 255–267, 2005.

[108] N. Ramsey, J. Dias, and S. Peyton Jones. Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. In *Proceedings of the Symposium on Haskell*, pages 121–134, 2010.

[109] D. Rémy and J. Vouillon. Objective ML: An Effective Object-Oriented Extension to ML. *Theory And Practice of Objects Systems*, 4(1):27–50, 1998.

[110] M. Ringwald and K. Römer. Deployment of Sensor Networks, Problems and Passive Inspection. In *Proceedings of the Workshop on Intelligent Solutions in Embedded Systems*, WISES, pages 180–193, 2007.

[111] M. Ringwald, K. Römer, and A. Vialetti. Passive Inspection of Sensor Networks. In *Proceedings of the Conference on Distributed Computing in Sensor Systems*, DCOSS, pages 205–222, 2007.

[112] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321.

[113] K. Römer and J. Ma. PDA: Passive Distributed Assertions for Sensor Networks. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 337–348, 2009.

[114] A. Ronacher. Jinja2 Documentation. `http://jinja.pocoo.org/docs/`.

[115] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[116] S. Rost and H. Balakrishnan. Memento: A Health Monitoring System for Wireless Sensor Networks. In *Proceedings of the Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, SECON, pages 575–584, 2006.

[117] K. Römer, P. Blum, and L. Meier. Time Synchronization and Calibration in Wireless Sensor Networks. In I. Stojmenovic, editor, *Handbook of Sensor Networks: Algorithms and Architectures*, pages 199–237. John Wiley & Sons, 2005.

[118] K. Römer and F. Mattern. The Design Space of Wireless Sensor Networks. *IEEE Wireless Communications*, 11(6):54–61, 2004.

[119] J. Sallai, M. Maróti, and Á. Lédeczi. A Concurrency Abstraction for Reliable Sensor Network Applications. In *Proceedings of the Conference on Reliable Systems on Unreliable Networked Platforms*, pages 143–160, 2007.

[120] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R Smith. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, 2008.

[121] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP), 2011. draft-ietf-core-coap.

[122] O. Shivers. Continuations and Threads: Expressing Machine Concurrency Directly in Advanced Languages. In *Proceedings of the SIGPLAN Workshop on Continuations*, 1997.

[123] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, and M. Welsh. Simulating the Power Consumption of Large-Scale Sensor Network Applications. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 188–200, 2004.

[124] G. Simon, M. Maróti, Á. Lédeczi, G. Balogh, B. Kusy, A. Nádas, G. Pap, J. Sallai, and K. Frampton. Sensor Network-Based Countersniper System. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 1–12, 2004.

[125] T. I. Sookoor, T. W. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Global Views of Distributed Program Execution. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 141–154, 2009.

[126] T. I. Sookoor, T. W. Hnat, and K. Whitehouse. Programming Cyber-Physical Systems with MacroLab. In *Proceedings of the Conference on Embedded Network Sensor Systems*, SenSys, pages 363–364, 2008.

[127] R. M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection. `http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/`.

[128] R. M. Stallman and Z. Weinberg. The C Preprocessor. `http://gcc.gnu.org/onlinedocs/gcc-4.7.2/cpp/`.

[129] R. Stallmann, R. Pesch, and S. Shebs. *Debugging with GDB*. GNU Press, 2011.

[130] H. Sutter and J. Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, 2005.

[131] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan. Aveksha: A Hardware-Software Approach for Non-Intrusive Tracing and Profiling of Wireless Embedded Systems. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 288–301, 2011.

[132] The Eclipse Foundation. Eclipse: A Multi-Language Software Development Environment. `http://www.eclipse.org`.

[133] The Python Software Foundation. The Python Language Reference. `http://docs.python.org/2/reference/`.

[134] The Santa Cruz Operation, Inc. and AT&T. System V Application Binary Interface. `http://www.sco.com/developers/devspecs/gabi41.pdf`, 1997.

[135] M. Torgersen. Asynchronous Programming in C# and Visual Basic. Technical Report 14058, Microsoft, 2010.

[136] N. Tsiftes, J. Eriksson, and A. Dunkels. Low-Power Wireless IPv6 Routing with ContikiRPL. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 406–407, 2010.

[137] UC San Diego. ARGO - Global Ocean Sensor Network. `http://www.argo.ucsd.edu`.

[138] R. von Behren, J. Condit, and E. Brewer. Why Events are a Bad Idea (for high-concurrency servers). In *Proceedings of the Conference on Hot Topics in Operating Systems*, HotOS, 2003. Article No. 4, 6 pp.

[139] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 268–281, 2003.

[140] P. Wadler. Comprehending Monads. In *Proceedings of the Conference on Lisp and Functional Programming*, LFP, pages 61–78, 1990.

[141] P. Wadler. The Essence of Functional Programming. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL, pages 1–14, 1992.

[142] M. Wand. Continuation-Based Multiprocessing. In *Conference Record of the Lisp Conference*, pages 19–29, 1980.

[143] B. Warneke, M. Last, Liebowitz B., and K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *Computer*, 34(1):44–51, 2001.

[144] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, NSDI, pages 29–42, 2004.

[145] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees. Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing*, 10(2):18–25, 2006.

[146] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. E. Culler. Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 416–423, 2006.

[147] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A Comprehensive Source-level Debugger for Wireless Sensor Networks. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 189–203, 2007.

[148] Y. Yao and J. Gehrke. The Cougar Approach to In-network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3):9–18, 2002.

[149] D. Zonta, H. Wu, M. Pozzi, P. Zanon, G. P. Ceriotti, M. Picco, A. L. Murphy, S. Guna, and M. Corrà. Wireless Sensor Networks for Permanent Health Monitoring of Historic Constructions. *Smart Structures and Systems*, 6(5):1–20, 2010.

[150] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-Level Sensor Network Simulation with COOJA. In *Proceedings of the Workshop on Practical Issues in Building Sensor Network Application*, SenseApp, pages 641–648, 2006.

List of Figures

List of Tables

List of Functions

List of Data Types

List of Abbreviations

API	application programming interface
ASR	abstract syntax representation
AST	abstract syntax tree
CLI	command line interface
CPP	GCC preprocessor
CPU	central processing unit
CSP	Communicating Sequential Processes
CSR	concrete syntax representation
DCA	data collection and in-network aggregation
ENI	enriched node info
FGL	Functional Graph Library
FQN	fully qualified name
GCC	GNU Compiler Collection
GDB/MI	GDB Machine Interface
GEN	generated variant
GHC	Glasgow Haskell Compiler
GUI	graphical user interface
IDE	integrated development environment
IO	input and output operation
IoT	Internet of Things
IR	intermediate representation
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
NAT	native variant
NI	node info

OS	operating system
PAL	platform abstraction layer
RAM	random access memory
ROM	non-volatile programmable memory
RPC	remote procedure call
SP	stack pointer register
TL	threaded variant
UNI	undefined node info
WSN	wireless sensor network

Index