An extension of embedded C for parallel programming

Master-Thesis von Bastian Gorholt
August 2014

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Software Engineering Group

An extension of embedded C for parallel programming

Vorgelegte Master-Thesis von Bastian Gorholt

1. Gutachten: Sebastian Erdweg
2. Gutachten:

Tag der Einreichung:

1 Introduction

- mbeddr is a language and ide for embedded programming that facilitates programming by providing higher level language mechanisms
- parallel programming is becoming ever more important
- yet, mbeddr is still lacking higher-level support for parallel programming
- compared to library extensions language based support for pp features multiple benefits: individual problem-specific syntax, reduction of erroneous input by type system enhancements, program-wide optimization of generated code
- short examples for benefits
- ParallelMbeddr introduces task-based explicit parallel programming with shared memory that can be synchronized
- Objectives: apropriate syntax, type system support for safer code, optimized output
- outline of the work

Kommentar von Bastian
Question: Introduce example that is later used for the evaluation already here for motivational reasons?

2 Background

2.1 Processes and Threads

Every program in (delayed or interrupted) execution is represented by a process. A process typically has its own protected data (via virtual memory) and execution state and consists of one or more threads. A thread, also known as lightweight process, shares some of the memory with its process but has its own execution state and thread-local storage as needed[**?**, p. 20]. In the course of programming language and operating system development several variants of threads have been devised, among others green threads, fibers and coroutines which mainly differ in how they are managed.

2.2 Parallelism and concurrency

If multiple threads are "in the middle of executing code at the same time"[14, p. 124(?)] they are processed concurrently. They can actually executed at the same time on different processors or interleaved on a single processor. The former is further called parallelism. Parallelism generalized to "the quality of occuring at the same time"[6, p. 91] can manifest in different ways. TODO: consider mentioning data races that are involved with both execution models

2.3 Types of parallelism

At least four different kinds of parallelism have been conceived. From an application programmer's view seen at a very low level of the computer exist bit-level parallelism and instruction-level parallelism. Bit-level parallelism is concerned with increasing the word size of processors in order to reduce the amount of cycles that are needed to perform an instruction[5, p. 15]. Instruction-level parallelism, also called pipelining, is the simultaneous utilization of multiple stages of the execution pipeline of the processor. Both bit-level parallelism and instruction-level parallelism primarily reside on the hardware level and the operating system level and are, thus, no subject of this work. Data parallelism is present if the same calculation is performed on multiple sets of data and can be regarded as a specialization of task parallelism which denotes the simultaneous execution of different calculations "on either the same or different data"[5, p. 125]. The latter being the more general approach to software-level parallelism is the subject of this work.

2.4 Data races

With multiple threads running in parallel and having access to the shared data of their process a second class of errors that is unique to parallel (and distruted) programming arises. These so-called synchronisation errors occur due to data races and are a result of the general non-atomicity of computations and memory references. Data races— also known as race conditions— are defined as at least "two unsynchronised memory references by two processes on one memory location, of which at least one reference is a write access"[7, p. 327][1]. Such data races can result in inconsistent program states and non-deterministic program behavior since the order in which the concerned memory is accessed might change. In order to deal with this issue three main paradigms have been conceived in parallel programming.

[1] As the definition implies data races are not limited to threads and the shared process data. E.g. file-based race conditions can even occur between two different processes[19]. Since this work is about parallelization of single processes other kinds of race conditions are not further considered.

2.5 Communication model: shared memory

The memory model that underlied the former treatment of processes that share some data with their threads is formally known as *shared memory* . In this model communication between entities is realized by shared-memory regions which are written to and read from[10, p. 138][2]. Data races can be avoided with help of the low-level the synchronization primitive *mutex*[3]. A mutex can be locked by exactly one thread. Any other thread that tries to lock the same mutex is blocked until the locking thread releases the mutex[15]. Thus, code regions can be protected by having threads synchronize over mutexes that protect these regions. One of the disadvantages of mutexes is that they are not tightly coupled to the data or computation that they protect. It is the programmer's duty to take care of the sane utilization of a certain mutex. Therefore various higher-level synchronization measures like monitors in Java(TODO: forward reference) and synchronized classes in D(TODO: forward reference) were developed. These measures are usually built on top of mutexes[13, p. 25]. Another disadvantage of mutexes is their vulnerability for deadlocks which means that multiple processes are in a state where "each is waiting for release of a resource which currently held by some other process"[2, p. 119] such that no progress will ever finish executing[8, p. 2-3].

2.6 Communication model: message passing

Whereas communication in the shared memory paradigm happens rather implicitly it is done explicitly in the *message passing* paradigm. Message passing originates from Hoare's paper on Communicating Sequential Processes (CSP)(TODO: reference!). In CSP messages are sent from one entity to another. "The sender waits until the receiver has accepted the message (*synchronous* message passing)"[18, p. 138]. Message passing with asynchronous message sents were deployed by the actor model[12] and pi calculus[16]. Although message passing avoids shared data and realizes communication generally via copies of data[4] it still suffers from potential race conditions[17].

2.7 Communication model: transactional memory

Transactional memory provides a non-blocking[5] memory model which enables communication via "lightweight, *in-memory* transactions"[9, p. 3] which are code blocks that from a programmer's perspective are executed atomically. The illusion of atomicity is realized by the underlying transaction system which may execute transactions in parallel and has to take care of conflicting reads and writes in transactions[6][11]. Transactional memory can either be realized in hardware or in software. While the former promises a better performance it demands specific hardware. Software transactional memory on the other hand seems to suffer from comparatively "poor performance"[3, p. 13].

2.8 Coarse- and fine-grained synchronization

In order to keep structures and computations synchronized the simplest approach to avoid data races is to use the available measures like locks or transactions as broadly as possible. E.g. transactions could be widened to hold every operation a thread has to execute. As every synchronization is basically a serialization of otherwise parallel executed code such coarse-grained synchronization would eliminate the benefits of parallel execution. On the other hand fine-grained synchronization can introduce race conditions if the programmer misses some locking policy. In addition the acquisition of every lock takes time which can become an issue with increasing locking counts. Therefore a trade-off between locking-overhead and scalability problems has to be found[9, pp. 1-2].

[2] As for race conditions the model is not limited to intra-process communication via threads or similar approaches to parallelization. Communication between two process can also be realized via shared memory but is not in the scope of this work.

[3] Semaphores as a second synchronization primitive are closely related to mutexes. Since they do not provide further insights for the discussion they are not further investigated in this work.

[4] Actually shared data is often used in implementations of message passing models in order to enhance the performance. Furthermore it exists on the language level like in terms of monitors that were developed by Hoare to reduce deadlocks in CSP. The main notion of the message passing concept nevertheless goes without shared data.

[5] TODO: explain

[6] To this end the corresponding transactions may need to be reexecuted as a whole.

2.9 Embedded programming

“An embedded system is a computerized system that is purpose-built for its application.”[21, p. 1] Due to its narrow scope and monetary constraints induced by the application domain the hardware of such systems is often constrained to the point that it just accomplishs the job[21]. Thus, the memory consumption of the resulting program is one main issue to be considered in embedded programming. Additionally, for real-time systems which constitute a subclass of embedded systems not only the correctness of computations but also the consumed time determine their quality and usefulness [4, pp. 1-2]. Therefore the predictability of the program’s execution time becomes an issue for real-time systems[7].

2.10 MPS and mbeddr

“JetBrains MPS[8] is an open source [...] language workbench developed over the last ten years by JetBrains. ”[20] As such it provides an projectional editor which lets the user directly work on the abstract syntax tree (AST) of the program[1]. It supports the development and composition of potentially syntactically ambiguous modular language extensions in combination with the development of integrated development environments or extensions thereof. Mbeddr is an extension of MPS tailored for the embedded software development in C. Every program written in the mbeddr IDE is translated to C99 source code which are then be further processed by the gcc tool chain[9]. The implementation of C in mbeddr does not only provide extensions to the core of C but also has a few differences to the basis of C99. They will be introduced as needed.

Kommentar von Bastian
Add explanations for MPS aspects, type-system checking rules... as needed

Kommentar von Bastian
Maybe add section for explicit vs. implicit parallel programming, look at Programming Distributed Systems by H. E. Bal, pp. 113-114

Kommentar von Bastian
Move to implementation: “In order to abstract from the actual implementation, e.g. as a thread, and generalize the language design a concurrent execution unit is called task. Similar to a thread a task may share data with its process.”

Kommentar von Bastian
Implementation: coarse- vs. fine-grained synchronization, problems => implicit synchronization not exhaustive => optimization for safe lock avoidance helpful

[7] Mbeddr(TODO: forward reference) does not have first-class support for the quantification of related parameters like the worst-case execution time (WCET)[4, p. 8], yet. For this reason predictability is a lesser concern of this thesis and will be reflected primarily in the careful consideration of the CPU consumption and processing time of the implementation.

[8] http://jetbrains.com/mps

[9] http://gcc.gnu.org/

3 Analysis

3.1 Existing work

Introduce parallel programming concepts of other languages and how they relate to this work.

3.2 Language Design

New syntax, typing rules and generation output for:

- tasks
- futures
- sychronization primitives

Kommentar von Bastian
Question: Where to put the reasoning behind each language construct?

4 Design and Implementation

In this chapter extension of mbeddr for parallel programming, called ParallelMbeddr, is introduced. To this end the new language features for C are explained each in terms of the design and the translation to plain mbeddr C code. Furthermore relevant implementation details are briefly depicted. A running example will help to illustrate the presented features. **Kommentar von Bastian**
TODO: explain running example

4.1 Tasks

The basic parallelization element is a *task*. It denotes a parallel unit of execution. Its named deliberately divergently from the prevalent parallelization terms(reference to basisc) in order to abstract from the concrete implementation which might change in the future.

4.1.1 Design

The syntax e of expressions in mbeddr is extended with

e ::= ... | |e|

When executed a task term yields a handle to a parallel unit of execution. This way the initialization of the task and the actual execution are decoupled and can happen . When it is run the embraced expression is executed and its value is returned. If the type of the expression is void no value will be returned. The type of a task reflects this return value.

t ::= ... | Task<t>

Due to implementation reasons the embraced return type of a task must be either void or a pointer to the type of the embraced expression:

e :- void ———————— |e| :- Task<void>

e :- t, t != void ———————— |e| :- Task<t*>

4.1.2 Translation

The pthreads library was chosen as a means to realize concurrency in the translation. It supports all necessary parallelization features and provides a more direct control of the generated code as opposed to frameworks like OpenMP (TODO: reference). Every task in ParalllelMbeddr is represented by a thread in pthreads. As the thread initialization function of pthreads takes a function pointer of type void* -> void* the computation of the translated task is represented by an according function:

void* parFun_X(void* voidArgs) ...

The X in the name symbolizes that for every task a unique adaptee of this function with the prefix parFun_ and some unique suffix chosen by the framework is generated.

If a task contains any references to local variables or function arguments they need to be bound to capture the variable state at the time of the task initiliazation. Such state is represented by an argument struct:

struct Args_X t_1 v_1; ... t_n v_n;

where every v_i denotes an equally named reference in the task expression of type t_i.

The generated function parFun_X is then given an instance of Args_X which it uses to bind the references of the task expression to. The full function definition of a task e of type Task<t*> is, thus:

void* parFun_X(void* voidArgs) t* result = malloc(sizeof(t)); Args_X* args = (Args_X*) voidArgs; *result = e'; return result;

where e' is the expression obtained when every local variable reference and function argument reference r in e is substituted by a reference to an equally named and typed field in args:

r / args->r

If the embraced expression of a task does not contain any reference of this kind (e.g. only references to global variables) the args definition line is omitted as is clearly the—otherwise empty—struct Args_X. In this case e' equals e.

The generated function of a Task<void> renounces the result-related statements:

void* parFun_X(void* voidArgs) t* result = malloc(sizeof(t)); Args_X* args = (Args_X*) voidArgs; e';

Again, any argument related code is generated as needed.

The aforementioned handle that a task yields is represented by an instance of a corresponding struct that captures both the initilization state and the computation of the embraced expression[10]:

struct Task void* args; (void*) => (void*) fun;

4.2 Tasks and Futures

for all items mentioned in the language design and necessary intermediary MPS concepts:

- implemented structure
- implemented typing rules
- implemented generation rules

4.3 Synchronization

see above

[10] (void*) => (void*) fun is mbeddr syntax for the not easily edible function pointer void *(*fun) (void *) in standard C99

5 Evaluation

- show if and how objectives mentioned in the introduction are met by the implementation
- use ilustrating examples

Literaturverzeichnis

[1] Language workbenches: The killer-app for domain specific languages?

[2] *Introduction To Operating Systems: Concepts And Practice An 2Nd Ed.* Prentice-Hall Of India Pvt. Limited, 2007.

[3] L.W. Baugh and University of Illinois at Urbana-Champaign. *Transactional Programmability and Performance*. University of Illinois at Urbana-Champaign, 2008.

[4] G.C. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency*. Series in Computer Science. Springer, 2006.

[5] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/software Approach*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Morgan Kaufmann Publishers, 1999.

[6] D.M. Dhamdhere. *Operating Systems: A Concept-based Approach,2E*. McGraw-Hill Higher Education, 2006.

[7] E.H. D'Hollander, G.R. Joubert, F. Peters, and U. Trottenberg. *Parallel Computing: Fundamentals, Applications and New Directions: Fundamentals, Applications and New Directions*. Advances in Parallel Computing. Elsevier Science, 1998.

[8] I.A. Dhotre. *Operating Systems*. Technical Publications, 2007.

[9] R. Guerraoui, M. Kapałka, and N. Lynch. *Principles of Transactional Memory*. Synthesis lectures on distributed computing theory. Morgan & Claypool Publishers, 2010.

[10] S. Haldar and A.A. Aravind. *Operating Systems*. Pearson Education, 2009.

[11] T. Harris, J.R. Larus, and R. Rajwar. *Transactional Memory*. Synthesis lectures in computer architecture. Morgan & Claypool, 2010.

[12] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[13] A. Holub. *Taming Java Threads*. Apresspod Series. Apress, 2000.

[14] B. Lewis and D.J. Berg. *Multithreaded Programming with Java Technology*. Sun Microsystems Press Java series. Prentice Hall, 2000.

[15] C.S. LLC, M.L. Mitchell, A. Samuel, and J. Oldham. *Advanced Linux Programming*. Pearson Education, 2001.

[16] Robin Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.

[17] Robert H. B. Netzer, Timothy W. Brennan, and Suresh K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '96, pages 31–40, New York, NY, USA, 1996. ACM.

[18] J. Protic, M. Tomasevic, and V. Milutinović. *Distributed Shared Memory: Concepts and Systems*. Systems Series. Wiley, 1998.

[19] C. Steven Hernandez. *Official (ISC)2 Guide to the CISSP CBK, Second Edition*. (ISC)2 Press. Taylor & Francis, 2009.

[20] Markus VÃ¶lter. Generic tools, specific languages.

[21] E. White. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media, 2011.