Tuning MetaOCaml Programs for High Performance

Diplomarbeit

Autor:
Tobias Langhammer

Aufgabensteller:
Prof. Christian Lengauer, Ph.D.
Lehrstuhl für Programmierung
Universität Passau

Betreuer:
Dr. Christoph Herrmann

Zweitgutachter:
Priv.Doz. Dr. Martin Griebl

eingereicht: 6. September 2005
revidiert: 24. Oktober 2005

Abstract

The domain of high-performance computing is still dominated by manual optimizations of programs written in C or Fortran. The reason why high-level languages failed to gain ground is the execution overhead entailed by the potential abstractions introduced.

This thesis proposes MetaOCaml for enriching the domain of high-performance computing by multi-staged programming. MetaOCaml extends the OCaml language by three new constructs in order to provide run-time code generation. By tagging OCaml code fragments with a pair of special brackets, they become abstract data objects of the program enclosing them. This program can create and combine code objects at run time as well as execute them in its environment by the application of a single operator. This allows the easy construction of speed-optimized code. Furthermore, like OCaml, MetaOCaml is a language with static types. Though code objects are generated and executed at run time, its type system guarantees the type correctness of run-time generated code at the time the program is compiled.

This thesis presents important speed-related issues, like speed-aware coding style, offshoring or interfacing to C. Three case studies demonstrate the applicability of these in realistic contexts and discuss the impact in terms of execution speedup.

Focusing on numerical applications, the first case study examines iterative and recursive implementations of matrix multiplications, comparing low-level C optimizations with high-level implementations using MetaOCaml. Hybrid implementations show how to combine the optimization opportunities of both approaches.

A second case study demonstrates how MetaOCaml can be used for the implementation of a domain-specific language. The presented compiler combines a simplifier working on the abstract syntax and a code generation function using MetaOCaml syntax. As the code generator can be constructed by adding MetaOCaml constructs to the code of an interpreter, the effort of this implementation is marginal compared to the speedups attained.

Finally, the case study of a parallel Karatsuba polynomial multiplication presents an embedded specification language for a static parallel computation structure. The implementation of this language gives an example of how specialized code can be generated with MetaOCaml by using information about the parallel run-time environment. Also, it demonstrates a clean approach to separate the concerns of the application implementer and the parallel engineer.

Benchmarks for all presented implementations confirm that the execution time can be reduced significantly by high-level optimizations. Some MetaOCaml programs even run as fast as respective C implementations. Furthermore, in situations where optimizations in pure MetaOCaml are limited, computation hotspots can be explicitly or implicitly exported to C. This combination of high-level and low-level techniques allows optimizations which cannot be obtained in pure C without enormous effort.

Acknowledgments

This diploma thesis was written as final years project at the Chair for Programming, University of Passau, Germany.

I would like to thank all staff members who helped me with their suggestions and time. Especially, I like to thank Christoph Herrmann for his substantial support and constant guidance and for giving me the opportunity of participating in current research. Special thanks to Christian Lengauer for his detailed suggestions and comments on the English version.

Many other people supported me in many different ways: Christof König, by his comments on the final draft, Timon Christl, by helping me with tricky software problems and Michael Classen, by sharing thoughts and coffee. Also, I like to thank my parents and family, who gave me the time and support I needed to complete my work, and my friends in Passau and Bayreuth, who where actually not understanding what I was doing but gave me precious distraction, here and then.

Contents

Chapter 1

Introduction

1.1 Motivation

To this day, Fortran and C are the most popular programming languages for implementing programs in the field of high-performance computing. This is primarily motivated by the ability of Fortran and C compilers to generate fast, optimized executables for almost any platform available. Another convenience of using these languages is the simple compilation process, which offers machine-independent programming on the one hand, and straightforward understanding of the assembly of the target program on the other. As a drawback, C and Fortran do not offer the level of abstraction and safety modern high-level languages do, e.g., the amenities of strong typing, automatic memory management and programming abstractions like object-orientation or unnamed functions.

This thesis presents approaches for bridging this gap between programming comfort and execution efficiency. The language of choice is *Objective Caml* (Caml Consortium, 2005), or *OCaml* for short, a multi-purpose programming language which provides a wide range of abstraction mechanisms as well as the fast execution of target programs. Unfortunately, typical OCaml programs cannot compete in performance with C programs, let alone with those aggressively optimized. This problem can be tackled by applying different techniques, which are addressed in detail in the course of this thesis.

Programming style. There are a number of recommendations, given by experienced developers and OCaml compiler implementers, for coding efficient OCaml programs. They are all founded on knowledge about the compilation process and the run-time behavior. E.g., in a certain context, a programmer has to know whether a function is being inlined or whether a recursive function is as fast as an iteration. Also the run-time system can be tuned to the needs of the run-time environment. E.g., the garbage collector can be adjusted to trade off between memory consumption and execution time.

Interfacing to low-level languages. Analysis or profiling of programs commonly

shows that most time-consuming calculations are restricted to certain small functions or fragments of code. Such hotspots typically originate from repeated calls in a loop or a recursion. To profit from the execution efficiency and the optimization capabilities offered by low-level languages, these fragments can be implemented as C functions or Fortran subroutines. The interfacing mechanism provided by OCaml makes them callable like any genuine OCaml function.

Runtime code generation. MetaOCaml (Taha et al., 2005) is an extension of the OCaml language, which provides a convenient way to generate and run OCaml code at run time. MetaOCaml introduces only three new constructs and, unlike other code generation tools, generates not only well-formed but also statically well-typed code.

The important field of application for MetaOCaml is the elimination of interpretation overhead. Instead of writing a subroutine which requires all arguments in order to perform its computation, the invocation and execution can be split into two stages: the first stage performs only those operations which depend exclusively on input available at a certain point time. For the remaining operations, OCaml code is generated and returned as a *residual program* awaiting the rest of the input. The second stage consists of the execution of this residual program. By splitting the computation in two stages, the running time of a repeated execution can be reduced significantly if the first stage requires only a single execution and the iteration is performed on the simpler second-stage code.

Further optimization techniques. Another way of speeding up OCaml code is to make use of existing C compilers by transforming computation-intensive code automatically to equivalent C code. Eckhardt et al. (2005) present a light-weight solution called *offshoring* by restricting the transformable constructs to an adequate subset of OCaml. Unlike the complex C interfacing mechanism provided by OCaml, the implementer simply tags code he plans to be offshored with special brackets and initiates the compilation and execution with a special run operator.

Unlike native machine code, bytecode represents programs in a machine-independent way and is executed on a virtual machine. While simple virtual machines perform an interpretation of the bytecode, *just-in-time (JIT) compilation* (Starynkevitch, 2004) is a technique which speeds up the execution by compiling bytecode dynamically at run time. A code fragment is compiled to native code before its first use. Thus, a speedup of the running program can be expected if the overhead of the compilation can be amortized by frequent execution of the compiled code.

1.2 Language Features of MetaOCaml

OCaml, which forms the basis of the MetaOCaml distribution, is a programming language which combines functional programming, derived from the ML programming

language, with constructs of the object-oriented programming paradigm. It was developed in the "Cristal Project" group at the INRIA Rocquencourt research institute, and is freely available for various platforms including Unix/Linux, Windows and MacOS.

OCaml has a static type system, which enhances the safety of compiled programs. It guarantees the absence of false data usage and incorrect access of compound data. Like in Standard ML, in many cases there is no need to provide explicit type information because correct types can be inferred by the compiler. Nevertheless, adding type information to the program is widely encouraged. It enables the compiler to support the programmer with more detailed information about type errors, and it enhances the readability of the code by providing additional documentation.

Typically for functional languages, functions are considered as "first class citizens", i.e., like any data object, functions can be created at any point of the program and can serve as argument or return value of another function. As this requires a rather complex memory management, OCaml features automatic garbage collection, so explicit memory management is not needed. The object-oriented extension integrated in OCaml obeys the strong type checking paradigm of the language, too.

Like in any modern programming language, OCaml programs can be modularized by separate compilation units and by linking them to an executable file. The compilation may yield a native program to be run on the desired platform or a bytecode executable to be run with a bytecode interpreter. The OCaml bytecode interpreter is a virtual stack machine, like the JAVA virtual machine, but, in contrast to JAVA, the OCaml bytecode also contains constructs for an effective creation and application of so-called *closures*, the compiler's representation of function objects.

1.3 Meta-Programming

In many cases, state-of-the-art software development already uses techniques to automate the creation of specialized code. The idea of meta-programming is to generate code by a *meta-program* in a first stage, and to run the residual program in a second stage. The advantages of this technique are an increase in maintainability of large projects, like autoconf and automake (The Free Software Foundation, 2005b), an improved reusability of code, like C++ templates (Lippman and Lajoie, 1998, Chap. 10 and 16), and, which is an important aspect of the subsequent examinations, the ability to construct optimized code from a high-level specification. Unfortunately, most programming languages still lack safe and convenient facilities for generating program code. One common way of forming valid source code is to use macro substitutions for combining code fragments. The drawback of this method, which is commonly used in C, is the permanent risk of producing syntactically incorrect or type-marred code. In C, we can define the macro

```
#define inc(n) (n++)
```

to increment the value of a variable n. While using `inc(`v`)` with any variable v produces valid code, the expansion of constants, like `inc(42)`, makes the compiler fail.

C++ templates remedy this problem only in part. While a template guarantees a syntactically well-formed expansion, its type correctness is not checked until the template is instantiated in the code (Czarnecki et al., 2004). This approach grants sufficient type safety if template expansion and compilation are performed in one run.

For a type-safe generation of code at run time, the static typing of code-generating functions has to be combined with the ability to construct and compile code at run time. Generating and compiling source code from strings does not address this problem because it carries the same risks of producing syntactically malformed code as C macros. Correct syntax can be guaranteed by constructing a parse tree of the target language via the meta-language, a method which also allows a subsequent inspection of the object. Unfortunately it does not guarantee the type correctness of the generated code.

1.3.1 Multi-staged Programming

As a means of run-time code generation, MetaOCaml introduces the concept of *multi-staged programming (MSP)* into the OCaml language. It extends the OCaml language by a pleasing syntax for constructing and running code at run time. Only three additional constructs have been added:

- Brackets (`.<` exp `>.`) for creating an object which represents the expression exp, the so-called *object code*.
- The unary escape operator (`.~`) for inserting object code into a new code object to be constructed.
- The unary run operator (`.!`) for compiling object code to an OCaml value.

As the nesting with code brackets may be arbitrarily deep, the base program code in MSP is called *first stage* and each new open bracket introduces a *new stage*. In terms of meta-programming, a program fragment of stage n always acts as meta-code for object code of stage $n + 1$. For an expression e of type b, the respective second stage code `.<` e `>.` is treated like a common OCaml object of type `('a,`b`) code`.[1]

Unlike other meta-programming systems, MetaOCaml does not provide a way to inspect code objects after their creation. Code inspection is neither supported nor encouraged because it would breach the static type system.

1.3.2 Example

To motivate the use of staged programming consider the example of a function computing the `n`-th power of a floating point number `a`.[2]

[1] The polymorphic type `'a` which comes along with the type of the code is called *classifier* (Calcagno et al., 2004). It is used by the type inference system for preventing the application of `.!` on code objects with variables which were not linked before. As classifiers are not explicitly used in an implementation using MetaOCaml constructs, they can be ignored by the user most of the time.

```
(* val power : float -> int -> float *)
let rec power a n =
  if n = 0 then 1.0
  else power a (n-1) *. a
```

Now, imagine a situation in which `n` is constant for a large number of calls of function `power`. Each invocation causes the execution of the whole recursion, though the number of recursive steps is known in advance. By using MetaOCaml, we are able to exploit this situation: a staged function takes the number `n` of recursive steps and unfolds the recursion to a closed expression forming a chain of multiplications.

```
(* val power : ('a,float) code -> int -> ('a, float) code *)
let rec power a n =
  if n = 0 then .< 1.0 >.
  else .< .~(power a (n-1)) *. .~a >.
```

Here, the staging is done by adding brackets where code fragments are to be constructed and by embedding recursively created code fragments with the escape operator (`.~`). Note also the change in the type signature: both the base value `a` and the return value of the power function become second-stage code objects. As it is used for unfolding, `n` remains a first-stage integer.

With this staged power function we still cannot generate a code object by giving only `n` as parameter. A wrapping function does the trick.

```
(* val powgen : int -> ('a, float -> float) code *)
let powgen n = .< fun a -> .~(power .<a>. n) >.
```

This function generates the complete code of the power function by embedding the chain of multiplications in the lambda expression.

Now, in combination with the run operator, we can create a specialized power function for $\mathtt{n} = 42$ at run time.

```
(* val pow_42 : float -> float *)
let pow_42 = .! (powgen 42)
```

Note what would happen if we stripped all staging annotations. The application of `powgen 42` would perform no unrolling but would simply yield a lambda expression expecting an argument `a`. As this example demonstrates, in case of termination, adding or removing staging annotations in pure functional programs affects only the order of execution and not the semantics.

1.4 Notation and Conventions

Throughout this thesis, fragments of code, like identifiers or types, are written in `teletype` font. For mathematical formulas the following notation is used. Let $n, m \in \mathbb{N}$, then

[2]Program variables, like any fragment of code, are written in `teletype` font. For details on the notation see Section 1.4.

$\mathbf{R}$ denotes a commutative ring $(R, +, \cdot)$ with neutral multiplicative element 1.

$n \operatorname{div} m$ denotes the integer division $\lfloor n/m \rfloor$.

$n \bmod m$ denotes the remainder of the division of n and m, i.e.

$$a \bmod b = a - b \cdot (a \operatorname{div} b).$$

$\overline{n}$ is an abbreviation[3] for the range $\{0, 1, \ldots, n-1\} \subset \mathbb{N}$.

B^A as well as notation $A \to B$ denote the set of mappings from set A to set B.

$f^{-1} \in A^B$ denotes the inverse of a bijective mapping $f \in B^A$, i.e.,

$$f^{-1}(y) = x \Leftrightarrow f(x) = y \qquad \text{for all } x \in A, y \in B.$$

$f|_S \in B^S$ denotes the function obtained by restricting the domain of $f \in B^A$ to $S \subseteq A$.

$\overline{n} \times \overline{m}$ abbreviates the Cartesian product

$$\{0, \ldots, n-1\} \times \{0, \ldots, m-1\}$$

$$= \{(i, j) | i \in \{0, \ldots, n-1\}, j \in \{0, \ldots, m-1\}\}.$$

$a \in A^{\overline{n}}$ denotes a finite mapping of natural numbers in $\{0, \ldots, n-1\}$ to elements of set A. Mapping a is isomorphic to an n-tuple with the i-th element given by $a(i)$. The subscript notation $a_i := a(i)$ acknowledges this fact. Tuple a is an appropriate modeling of an array of size n consisting of type A entries.

$M \in A^{\overline{n} \times \overline{m}}$ denotes a mapping of elements in $\{0, \ldots, n-1\} \times \{0, \ldots, m-1\}$ to elements in set A. Mapping M is isomorphic to a matrix of size $n \times m$ containing values of M. The subscript notation $M_{i,j} := M(i, j)$ acknowledges this fact.

$w(M)$ and $h(M)$ denote the width and height of matrix $M \in A^{\overline{h(M)} \times \overline{w(M)}}$.

$b \in \{0, 1\}^{\overline{n}}$ denotes a mapping of elements in $\{0, \ldots, n-1\}$ to 0 or 1. Mapping b is an appropriate modeling of a bitmap of length n. To describe all entries of b, the notation $\langle b_{n-1} \ldots b_1 b_0 \rangle$ is used.

$a \hat{} b$ denotes the concatenation of bitmaps a and b.

$\#b$ denotes the length of bitmap b

$\mathcal{B}_n(k)$ denotes the bitmap representation of $k \in \mathbb{N}$ using n bits, i.e.,

$$\mathcal{B}_n(k)_i := (k \operatorname{div} 2^i) \bmod 2 \qquad \text{for all } i \in n.$$

$\mathcal{B}^{-1}(b)$ denotes the natural number by interpreting $b \in \{0,1\}^{\overline{n}}$ as binary number, i.e.,

$$\mathcal{B}^{-1}(b) := \sum_{i=0}^{n-1} b_i \cdot 2^i.$$

Note that for all $n > 0$

$$\mathcal{B}_n^{-1} = \mathcal{B}^{-1}|_{\{0,1\}^{\overline{n}}}$$

$a \bowtie b$ denotes the bitwise interleaving, i.e., for a and $b \in \{0,1\}^{\overline{n}}$,

$$(a \bowtie b)_i := \begin{cases} a_{i \operatorname{div} 2} & \text{if } i \text{ is odd} \\ b_{i \operatorname{div} 2} & \text{if } i \text{ is even} \end{cases} \qquad \forall i \in \overline{2 \cdot n}.$$

In order to simplify the notation of mathematical expressions, the following order of precedence applies to operations if parentheses are omitted. The precedences are given as list from highest to lowest. For non-associative infix operations the respective associativity is given in parentheses.

- power a^b (right).
- logarithm $\log_a b$.
- division a/b (left) and multiplication $a \cdot b$.
- subtraction $a - b$ (left) and summation $a + b$.
- fraction $\frac{a}{b}$

Furthermore, concatenation $a\hat{}b$ has a higher precedence than interleaving $a \bowtie b$.

1.5 Overview

In the course of this thesis we examine the impact of MetaOCaml as tool for writing speed-optimized programs.

For this purpose, Chapter 2 starts with the introduction of basic optimization techniques and tools which are universally applicable in many contexts. We show how to tune the garbage collector and what precautions have to be taken in order to access arrays or float values efficiently. We also give some advice on what coding style to be used for a given problem. Benchmarks of exemplary implementations provide a means of rating the applicability of these optimizations in a given context. We give

[3] This notation resembles the definition of natural numbers in set theory (Hrbacek and Jech, 1999, Chapter 3), where $0 := \{\}$ and $n := \{0, \ldots, n-1\}$ for $n \neq 0$.

a demonstration of staged programming for generating optimized code at run time and a short introduction to offshoring, which provides the automatic transformation of typed OCaml code fragments to C or Fortran. The latter technique is very powerful because it can keep up with pure C in terms of execution speed and definitely beats C in terms of comfort and safety. The chapter concludes with a short assessment of the current implementation of JIT compilation, which turned out not to be competitive in a high-performance context.

The main part of the thesis concentrates on three case studies demonstrating the wide-ranging applicability of optimizations in MetaOCaml. Benchmarks conclude these case studies, exhibiting the speedups for each implementation.

Chapter 3 demonstrates a number of alternative implementations of a matrix multiplication and compares them with exemplary implementations in C. Chapter 4 concentrates on a key feature of MetaOCaml, the generation of code guided by run-time information. The example presented shows that MetaOCaml is a useful tool for implementing an optimizing compiler for an abstract specification language. Finally, in Chapter 5 we see a demonstration how MetaOCaml simplifies the implementation of an embedded domain-specific language, which provides the description of static parallel computation structures.

Some practical details on the development environment are given in Chapter 6, which present the compilers and tools as well as the interfacing mechanism for integrating C modules in OCaml.

Chapter 7 concludes the thesis with an overview of all discussed optimizations. A table gives a concise survey of these techniques and advice on the applicability of each of them.

Chapter 2

Optimization Approaches

A number of coding guidelines for writing execution-efficient OCaml code are given by experienced OCaml programmers and compiler implementers (Leroy, 2002). This chapter summarize these guidelines and elaborate on their relevance by discussing the benchmarks of exemplary implementations. As OCaml is the basis of MetaOCaml, these guidelines are directly applicable to both languages. A demonstration of using MetaOCaml for loop optimization, a short discussion on the offshoring optimization and an assessment of JIT compilation conclude this survey of optimization techniques.

2.1 Tuning the Garbage Collection

Due to the observation that memory allocated longer ago is less likely to be freed than memory allocated recently, OCaml comes with a *generational garbage collection* (Chailloux et al., 2000, Chap. 9). As Grune et al. (2000) pointed out, generational garbage collectors are the fastest and most efficient garbage collections known. In the case of OCaml, it works in two stages: a frequent *minor collection* of memory chunks tagged to be new and an infrequent but complete and time-consuming *major collection* of all allocations. If new memory survives the first phase it is tagged to be old.

Because the major garbage collection is rather time-consuming, it is not initiated until a certain amount of memory is wasted. The space overhead variable in the control record of the garbage collector defines this amount relative to the amount of live memory on the heap. Therefore, the smaller this value the more CPU time is required to clean up the heap. In the current release 3.08 of OCaml the default value is set to 80 %, but in some cases it may be reasonable to set it to 100 % or even 200 %, depending on the program behavior and the amount of memory available.

2.2 Optimizing OCaml Arrays

OCaml provides the polymorphic data type `'a array` as the standard data type for arrays. Like most languages with arrays as basic linear data type, OCaml has a pleas-

operation	construct	description
create	`Array.make n i`	creates an array of length `n` and initializes all entries with `i`
	`[| v1 ; v2; ...|]`	creates an array with fixed size and entries given by values $v_1, v_2, \ldots$
read	`a.(i)`	reads entry `i` of array `a`
	`Array.get a i`	equivalent to `a.(i)`
	`Array.unsafe_get a i`	equivalent to `a.(i)` without bound checks
write	`a.(i) <- v`	writes value `v` to entry `i` of array `a`
	`Array.set a i v`	equivalent to `a.(i) <- v`
	`Array.unsafe_set a i v`	equivalent to `a.(i) <- v` without bound checks

Table 2.1: Important constructs and functions for OCaml arrays.

ing syntax for creating them and for reading and writing array entries. Additionally, the `Array` module in the OCaml standard library offers high-level functions for creating and manipulating arrays, some of which, like `Array.map` or `Array.fold_left`, resemble the high-level combinators on recursive lists. Table 2.1 shows the most important operations needed to create and access arrays in OCaml.

2.2.1 Effect of Boundary Checks

Like for JAVA arrays or C++ collections, the semantics of OCaml specifies that each array access is preceded by a boundary check. For any access `a.(i)` which does not satisfy the restriction `(i < Array.length a && i >= 0)`, a corresponding exception is raised.

To illustrate the effect of boundary checks, consider the following example.

```
  for i = 1 to Array.length a - 2 do
    a.(i) <- a.(i-1) +. a.(i) +. a.(i+1) /. 3.0
3 done
```

For each of w iterations for the loop 4 bound checks are required, which makes a total of $4 \cdot w$. At a closer look, actually, most of these checks seem superfluous as a sufficient check merely needs to evaluate the array accesses for both loop bound values. Unfortunately, OCaml does not perform an optimization by omitting any of these checks.

The simplest ways to eliminate boundary checks are either to use the command line option `-unsafe` of the OCaml bytecode or native code compiler or to use the functions `Array.unsafe_get` or `Array.unsafe_set`. As the name already suggests, the implementer is highly discouraged from using these options. Out-of-range indexing gets obscured, as there is no guarantee that the program stops or crashes in such a situation.

2.2.2 Optimizing Numerical Computations

One of the most profitable optimizations on arrays can be performed by eliminating polymorphism. Accessing arrays of polymorphic type always involves the determination of the actual type at run time. For monomorphic arrays, the type is known at

compile time, so the OCaml compiler can inline the respective read or write access in the calling code fragment or can choose an efficient data layout.

Furthermore, concerning numerical applications, both the bytecode and the native-code OCaml compiler offer a special treatment for expressions containing int and float values and a special internal representation for int and float arrays. Almost all other values have a *boxed* representation, i.e., they are kept within an allocated chunk of memory on the heap. As a consequence, they are subject to the garbage collection and accessing them always involves a dereferencing operation. Int values are never boxed, and float values are unboxed in certain situations.

If float values were always boxed, the costs for evaluating complex floating point expressions would be unacceptable for numerical computations. Therefore, the OCaml compilers suspends the boxing of float values in the following situations:

- The float value is the result of an arithmetic operation (`+.`, `-.`, `*.`, `/.`, `**`, `sin`, `cos`, `exp`, `log`, ...) that is used immediately by another arithmetic operation.
- The float value is a `let` bound of an unboxed value, which is exclusively used within arithmetic expressions. E.g.,

```
let a = b  *. c in a *. a         (* a unboxed *)
let a = b  *. c in a *. a *. f a  (* a boxed *)
```

- The float value is an element of an array or a component of a record which has only float components. In this case, all values are stored consecutively like in C arrays.
- The access of float arrays if it is done in the context of an arithmetic float operation. E.g., `a.(i) <- 4.5 *. a.(i)` performs one float read and one float store.

Note that in all other cases float values are treated as boxed. As a consequence, iteration should be preferred over recursion because function parameters of type float are passed boxed. Also records with two float entries should be preferred over tuples, e.g., for representing complex numbers, because the access of a tuple requires an additional dereferencing to unbox the float value.

2.3 Speed-aware Coding Style

A number of benchmarks were run on a 1 GHz Dual-Pentium III machine with 512 MB memory running Fedora Core 1 Linux as operating system. The results are displayed in Table 2.2 on the following page. The MetaOCaml version used was 3.08 alpha 023. For the timings the respective functions provided by the `Trx` module of MetaOCaml were used. These functions eliminate short-term caching effects by iterating the sample function sufficiently often to get a reliable timing. Also, the execution times displayed in Table 2.2 on the next page are the arithmetic mean of five independent runs of each benchmark in the benchmark suite.

style	implementation	running time in msec. bytecode GC memory overhead 80 %	bytecode GC memory overhead 200 %	native GC memory overhead 80 %	native GC memory overhead 200 %
list reduction length 2^{19}	`sumlength_rec`	443.74	440.26	131.55	131.50
	`sumlength_fold`	495.32	495.01	146.28	145.27
	`sumlength_loop`	423.13	423.68	127.63	127.38
array reduction length 2^{19}	`sumlength_array`	471.43	471.92	92.64	92.23
	`sumlength_array`*	467.35	466.16	91.97	91.63
parameter passing 30000 iter.	`curried`	4.62	4.62	0.12	0.12
	`uncurried`	8.11	8.11	0.12	0.12
argument factoring 30000 iter.	`no_factoring`	10.74	10.74	0.84	0.84
	`factoring`	9.87	9.86	1.10	1.10

* boundary checks disabled

Table 2.2: Benchmarks on 1GHz PIII, MetaOCaml 3.08 alpha 023, for different OCaml coding styles

2.3.1 Recursion vs. Iteration

In most observed cases, loop iteration in OCaml is faster than recursion. However, recursion is implemented quite efficiently and there are situations where it is as fast or even slightly faster than an equivalent implementation by a `for` or `while` loop. This applies especially for the traversal and construction of recursively defined data structures. A loop iteration would have to maintain a reference to such a structure and perform iterated de-referencings and assignments. These assignments of heap allocated data require the generational garbage collector to take additional precautions (Chailloux et al., 2000, chap. 9): an up-to-date table is kept to hold references from the old to the new generation. Using a purely functional, recursive function, this table of references need not be updated because younger memory chunks of the data structure created always reference older ones.

Where possible, functions should be expressed by tail recursion to help the compiler optimize the execution. In tail recursive functions the value returned by a recursive sub-call is not used for any further calculation. Therefore, in order to avoid the repeated allocation of memory, the recursive call can reuse all memory allocated in the previous call. The OCaml bytecode provides a special instruction for this purpose.

The following function implements a combined counting and summation of list entries using a tail recursion.

```
  (* val sumlength_rec : float list -> float * int *)
  let sumlength_rec xs =
3   let rec r xs (s,l) = match xs with
    | hd::tl -> r tl (s +. hd, l + 1)
    | [] -> (s,l)
    in r xs (0.0,0)
```

Note the technique of processing intermediate results in tail recursive functions: tuple `(s,l)` is updated and passed as argument to the next recursion rather than summed

up as return value of the recursive call.

As recursive function definitions often lack readability, in functional programming the use of high-level combinators is highly encouraged. These combinators also provide an abstraction from the computation, so the compiler would be able to choose a more appropriate or more optimized implementation. The functionality of the previous function can, thus, be expressed by a single fold reduction

```
  (* val sumlength_fold : float list -> float * int *)
  let sumlength_fold xs =
    let step (s, l) i = (s +. i, l + 1) in
4   List.fold_left step (0.0, 0) xs
```

The recursion is performed implicitly by `List.fold_left`, which keeps the intermediate result in a pair of size and length.

As can be seen from the benchmark results, whether it is the bytecode or native code compilation, the implementation using a fold reduction is about 10 % slower than the explicit recursion.

Though a recursive implementation may be favorable in terms of readability, it may not yield the performance one would expect from an iteration by a for loop. By using references to point to the list and to accumulate intermediate results, the same functionality can also be implemented by traversing the list with a loop.

```
1 (* val sumlength_loop : float list -> float * int *)
  let sumlength_loop xs =
    let s = ref 0.0 in
    let l = ref 0 in
    let xsref = ref xs in
6   while !xsref != [] do
      l := !l + 1;
      match !xsref with
      | (hd :: tl) ->
        s := !s +. hd;
11      xsref := tl;
      | _ -> failwith "impossible"
    done;
    (!s,!l)
```

The benchmark results in Table 2.2 on the facing page show that this implementation is only slightly faster than the previous implementations, though it is less understandable for a human reader.

One of the reasons why the imperative function using loops is not so easy to comprehend is due to the recursive nature of the list type. The “natural” type in the context of loops would be an array, because it provides random access and need not be deconstructed in order to access an element. An equivalent implementation would look as follows.

```
1 (* val sumlength_array : float array -> float * int *)
  let sumlength_array xs =
    let s,l = ref 0.0, ref 0 in
    for i = 0 to Array.length xs - 1 do
      s := !s +. xs.(i);
6     l := !l + 1;
    done;
    (s,l)
```

Actually, counting the number of array entries is superfluous as the array length can be looked up with `Array.length`, requiring a constant number of operations. Nevertheless, looking at the benchmark results for the native compilation, this function outperforms all other implementations in terms of execution time. This gain of time should not surprise if we remember the compiler optimizations noted in the section on OCaml arrays. Because `xs` is a monomorphic array of floats, the compiler uses a special representation of unboxed array elements arranged consecutively in memory. Furthermore, all intermediate results of the floating-point operations in line 5 are unboxed float values. Additionally, though the `for` loop contains only one array access, we can notice a small speed gain if an unsafe array subscript omitting the boundary check is used. For an implementation of a matrix multiplication this gain can be expected to be more pronounced, as the benchmarks of the subsequent case study in Section 3.5 show. Here, the array accesses are even more dominant compared to other computations in the loop body.

2.3.2 Currying Functions

Curried parameters passing is usually faster than passing tuples. When arguments are passed to a curried function, the OCaml compiler ensures that no extra memory is allocated on the heap. Actually, in OCaml an uncurried function always has one tuple as single argument. As a consequence, a tuple is created in almost all cases in which an uncurried function is applied to multiple arguments. This means that, for each invocation, space is allocated on the heap, which has to be freed by the garbage collection later on. The native code compiler usually optimizes named, uncurried functions such that parameters are passed via registers and no extra memory has to be allocated. The only case in which the bytecode compiler does not allocate extra space is the passing of an already constructed tuple to an uncurried function.

To demonstrate the negative effect of passing tuples, consider the following ternary functions, which are called to perform 30000 tail-recursive calls.

```
  (* val curried : float -> float -> int -> float *)
2 let rec curried a b n =
    if n = 0 then a +. b
    else curried b a (n - 1)

  (* val uncurried : float * float * int -> float *)
7 let rec uncurried (a,b,n) =
    if n = 0 then a +. b
    else uncurried (b, a, (n - 1))
```

The benchmark results illustrate the influence of the optimization performed by the compiler. The bytecode run exhibits a substantial slowdown of the uncurried function compared to its curried version. This gap closes when both functions are run as native code. Here the argument passing via registers annihilates the negative effect of passing tuples.

2.3.3 Factoring out Recursion Parameters

For a recursive function which takes a parameter that is constant throughout the evaluation of the function, a naïve implementation could pass the actual argument unchanged at the invocation of the next level of recursion.

```
(* val no_factoring : float -> float -> float -> float *)
let rec no_factoring a b c =
if c > 30000.0 then
  a +. b +. c
else
  no_factoring a b (c +. a +. b)
```

This example shows a tail recursion whose parameters a and b are passed unchanged. Aiming at the reduction of the overhead for passing parameters, the same functionality could be implemented by *factoring out* parameters a and b, i.e., by making them variables of the the same scope which encloses the recursive function.

```
(* val factoring : float -> float -> float -> float *)
let factoring a b c =
  let rec f c =
    if c > 30000.0 then
      a +. b +. c
    else
      f (c +. a +. b)
  in
  f c
```

Another name used for this technique is *lambda dropping* (Danvy and Schultz, 2000).

As the benchmarks show, in the case of the bytecode compilation the effect of this optimization is noticeable but not big. As OCaml owes its roots to functional programming, it puts much effort into minimizing the overhead of parameter passing. One of these optimizations is to keep parameters in registers, as long as some are available. In the native code generated for function `no_factoring` and factoring the situation is different. For both implementations the recursion is replaced by a loop iteration. In the case of the simpler function without factoring the compiler is able to keep more registers in memory. Apparently, this is not a principle problem because the compilation of the same program with an earlier version of MetaOCaml basing on OCaml 3.07 resulted in a smarter use of registers.

2.3.4 Limited Automatic Optimizations

One rule of thumb for programming with OCaml is not to expect the compiler to do high-level optimizations. For example, OCaml does not unroll loops to reduce the

number of loop bound checks. In contrast, Section 2.4 shows that MetaOCaml constructs are particularly suitable to generate an unrolled loop at a high level. Unlike automatic unrolling at compile time, the generation of code can benefit from run-time information, like the size of the instruction cache.

Another optimization which the OCaml compiler hardly takes into account is the inlining of small functions. Only the native code compiler performs this optimization for non-recursive functions in a fairly conservative way. Whenever execution speed is a key issue, a good strategy would be to perform the inlining of small functions by hand, or to generate inlined code with MetaOCaml.

2.4 Loop Optimization with MetaOCaml

The main field of application for MetaOCaml is the generation of speed optimized code, exploiting run-time information. An in-depth discussion on optimizing code transformations using a multi-staged evaluation is given by Cohen et al. (2005).

As most time-consuming calculations take place in loops, they are a very promising subject for optimizations. The technique of unfolding is especially lucrative for loops in which boundary checks and conditional jumps dominate the actual calculation of the loop body.

Unfortunately, a complete unfolding of loops is either impossible, because the actual loop bounds may be unknown at loop construction time, or not favorable because a complete unrolling would lead to code explosion. The simple way to unroll the following loop.

```
let x = ref 1 in
for i = 1 to n do
  x := !x * i
done
```

is to transform it to the semantically equivalent loops, where `step` is the number of instances of the loop body to be unrolled:

```
let x = ref 1 in
let i = ref 1 do
while !i <= n - step + 1 do
  x := !x * !i;
  x := !x * (!i+1);
  x := !x * (!i+2);
  ...
  i := !i + step
done
while !i <= n do
  x := !x * !i;
  i := !i + 1
done
```

The value of `step` has to be chosen appropriately to make the resulting code fit into the instruction cache. The replicated body is now embedded in a while loop with exactly (`n` div `step`) iterations. The second while loop is needed to perform the remaining `n` mod `step` computations.

To implement the unfolding using MetaOCaml, we first define a combinator to duplicate the loop body for `n` iterations.

```
  (* val unroll :
2    int -> (('a, int) code -> ('b, 'c) code) -> ('a, int) code ->
     ('b, 'c) code
   *)
  let rec unroll n body ix =
    if n = 1 then
7     body ix
    else begin
      let n' = pred n in
      .< ( .~(unroll n' body ix) ;
           .~(body .< .~ix + n' >.)  ) >.
12  end
  in
```

Note that the parameter `body` is itself a function which takes the code for the index as input and generates code for the loop body. As the escape operator (`.~`) inserts object code into object code within the first stage of the meta-code, the enclosing expressions are executed immediately when `unroll` is called. Thus, the code fragments for the body are generated and concatenated with the sequence operator (`;`) at each recursive step of `unroll` (lines 6 and 7).

Function `genfor` finally generates the partially unrolled code, exactly as outlined before.

```
  (* val genfor :
2    (('a, int) code -> ('a, 'b) code) -> int ->
     ('a, int -> int -> unit) code *)
  let genfor body step =
    .< fun lb ub ->
      let i = ref lb in
7     while !i <= (ub - step) + 1 do
        .~(unroll step body .< !i >.); i := !i + step
      done;
      while !i <= ub do
        .~(body .< !i >.); i := !i + 1;
12    done >.
  in
```

To make use of unrolling, the loop can now be written in four steps: defining the loop body, generating the loop code, compiling the code to OCaml and running the loop by applying the loop bounds.

```
  let x = ref 1 in
2 let body i = .< x := !x * .~i >. in
  let loop_code = genfor body step in
  let loop = .! loop_code in
  loop 1 n
```

Note that the number `step` of body replications is information that need not be known until the program is run. The appropriate value could be given to the executable as command line argument, according to a cost model or depending on preceding performance tests.

Several example application for MetaOCaml code generation are presented in the case studies of the following chapters.

2.5 Offshoring

Currently, specialized compilers exist for various application domains, like numerical computing or embedded systems. Furthermore, there are widely used multi-purpose compilers, like the GNU C Compiler, which benefited greatly from the sustained work of experienced developers. Aiming at making use of the optimization technology implemented in these compilers, offshoring is another multi-staged approach for generating efficient programs.

In order to describe and motivate offshoring, consider the basic idea of *explicit heterogeneous multi-staged programming* (Sheard, 2001), which aims to generate code objects in a desired target language, to translate them with a specialized compiler and to link them dynamically to the meta-program. The term heterogeneous denotes the fact that, unlike MetaOCaml, the language of the first stage is different from that of the second stage. Compared to a *offshoring* translation of OCaml code objects to an arbitrary target

- An increase of new constructs to suite the needs of the target language.
- The lack of generality, as these constructs correspond to constructs of the target language.
- The need for static typing of the generated programs.

Eckhardt et al. (2005) show how these issues can be addressed by *implicit heterogeneous multi-staged programming*, which incorporates an automatic *offshoring* translation of OCaml code objects to the the desired target language. Their implementation, which is part of the MetaOCaml distribution since version 3.08.0 alpha 020, currently supports C and Fortran as target languages. Offshoring restricts the syntax of both OCaml and the target language to respective subsets in which each construct in one language has an equivalent construct in the other. Thus, an OCaml code fragment restricted to this subset can easily be translated to the target language by a one-to-one mapping of each OCaml construct in this subset.

Note that the syntax restriction of the target code makes many optimizations impossible which could be done with explicit heterogenous meta-programs or by using the OCaml interfacing mechanism to C.[1]For example, the offshoring mechanism does not allow to opt for the more cache-friendly allocation of C arrays described in the subsequent case study. Nevertheless, the advantage of implicit heterogeneous multi-staged programming lies in the hidden translation, compilation and dynamic linking.

To summarize also the advantages of offshoring compared to the homogeneous staged programming of MetaOCaml:

- Like MetaOCaml, offshoring is homogeneous in the sense that it has static typing and introduces only few new constructs to the host language.
- Like MetaOCaml, the compilation of object code is implicitly done by a single run operator.

[1]See Section 6.2 for details on the OCaml interfacing mechanism.

- Unlike MetaOCaml, the compilation of object code is not performed by reusing the base language but by translation to C/Fortran, compilation with a C/Fortran compiler and by dynamic linking.
- The translation to C is a simple substitution of equivalent OCaml constructs, so the application developer has a precise perception of the generated C code.
- The application implementer benefits from optimizations implemented in widespread C and Fortran compilers, like the icc (Intel® Compilers) or gcc.

Section 3.4.2 presents a first example in the context of the matrix multiplication case study. There, we show the important use of offshoring in the optimization of computation intensive loop nests, hotspots which are commonly found in numerical computations.

2.6 Just-in-time Compilation

Just-in-time (JIT) compilation speeds up the execution of OCaml bytecode programs by replacing the bytecode interpreter with a run-time compiler. No modification of the bytecode executable is needed as the JIT compiler expects exactly the same virtual machine code.

The JIT compiler developed by Starynkevitch (2004) consists of a main translation loop containing a big switch statement whose cases match on bytecode instructions. For each case, corresponding machine code is constructed using the C macros of the Lightning code generation library (The Free Software Foundation, 2005a). This library provides code generation for a number of architectures, so JIT compilation works for platforms like i386-Linux/Windows or PowerPC-MacOSX machines.

In order to retain compatibility, the compiler has to keep the bytecode for the whole run in order to use its addresses. For the execution of a bytecode segment, an incremental compilation is invoked, translating each segment the first time of its use. The resulting machine code gets associated with the respective bytecode via a hash table mapping bytecode addresses to addresses of native code fragments.

The implementation of the JIT compilation was not investigated further in the subsequent case studies of this thesis. On the one hand, the current release was too immature to run reliably with applications like the image processor in the second case study. On the other hand, selected execution tests of the case study implementations could compete neither with a native MetaOCaml compilation nor with offshoring optimizations.

Nevertheless, the development of a fast JIT compiler is a realistic option. It still keeps the compiled code independent from the actual platform it is run. Furthermore, the primary intention of the development was to provide a replacement for bytecode interpreter `ocamlrun` of the underlying OCaml environment. So the community interested in further development can be expected to be much bigger than that of the users of the native MetaOCaml compiler. At the same time, the current implementation of

the JIT compilation is compatible not only with the OCaml bytecode interpreter but also with that of MetaOCaml.

Chapter 3

Case Study: Matrix Multiplication

Matrix multiplication plays a key role in many numerical applications and optimizing them is, therefore, an important issue. Matrix multiplication is typically defined as follows.

3.1 Definition (Matrix Product)
For two matrices $A \in \mathbf{R}^{\overline{u} \times \overline{v}}$ and $B \in \mathbf{R}^{\overline{v} \times \overline{w}}$, the matrix product $C = A \cdot B$ is defined by

$$C_{i,j} = \sum_{k=0}^{v-1} A_{i,k} \cdot B_{k,j} \qquad \forall i \in \overline{u}, j \in \overline{w}$$

□

A direct implementation of this definition yields a program which mainly consists of three nested loops: two for iterating over rows and columns of C and one for the inner summation, which corresponds to a dot-product of the i-th row of A and the j-th column of B. The algorithm has a running time of $\Theta(uvw)$, or, for square matrices of width N, a running time of $\Theta(N^3)$. Other algorithms, like the matrix multiplication by Strassen (1969), perform slightly better in terms of running time complexity but worse in terms of memory consumption. Nevertheless, matrix multiplication often dominates other program parts, like scalar operations or reductions, which has an execution time that is at most linear in N.

This section discusses a number of multiplication algorithms and shows corresponding implementations in MetaOCaml. Furthermore, it demonstrates how to make use of the knowledge presented in the previous chapters in order to accelerate each of these implementations. To compare the effect of these optimizations, a number of matrix multiplications are also implemented in pure C. An extensive series of benchmarks of the implementations presented concludes this chapter. They demonstrate in which cases MetaOCaml can compete with C, not only in terms of running time but also in terms of programming safety and abstraction.

3.1 Implementation Considerations

In order to implement all variants in a way which makes them comparable, the following decisions about the overall design of the benchmark suite were made.

1. All implementations use the same data type for representing the input matrices A and B and the output matrix C.
2. Since each implementation requires a matrix representation of its own, a pre- and post-processing stage is needed for the necessary conversions and run-time optimizations, like the generation of optimized code.
3. The benchmarks are the same for all implementations, i.e., both the pre- and post-processing stage as well as the actual matrix multiplication are implemented as separate program units and are benchmarked separately.
4. Each matrix has a square shape and a width which is a power of 2. More formally: the shape of each matrix M must satisfy the condition

$$\exists n \in \mathbb{N} : h(M) = w(M) = 2^n. \tag{3.1}$$

Common Data Representation in C Implementations

The default matrix representation used for C implementations is a dynamic double array, which, in effect, is a `double` pointer referencing consecutive memory which is allocated dynamically. The entries of this array are stored in row-major order, i.e., for row i and column j the value $A_{i,j}$ is stored at array position $i \cdot w(A) + j$.

As the dynamic allocation of memory for C arrays is done by an explicit function call yielding a pointer to a consecutive chunk of memory, an additional optimization can be achieved in order to reduce the number of page misses in the data cache. The optimization is based on the internal structure of the caching mechanism: the content of the cache is composed of *cache lines*, which are consecutive chunks of memory of a machine dependent, fixed size. The main memory is also divided into consecutive sections, which are all of the same length as a cache line. In order to accelerate the access to the main memory, the cache lines are filled with a number of these main-memory sections. Whenever an inquired date cannot be found in the cache (*cache miss*), a cache line gets refilled completely with the section of the main memory containing this date.

As Thiyagalingam et al. (2003) showed, the number of cache misses can be reduced considerably by allocating each array at a start address of a cache line section in the main memory. For this purpose, the POSIX standard (The Open Group, 2003) provides the special C function `posix_memalign()`, which is guaranteed to be compatible to `free()`, the standard C function for freeing previously allocated memory. The POSIX allocation function provides a parameter for an alignment for the allocation of memory. After a successful call, the first argument is set to a memory address which is a multiple of this alignment.

Common Data Representation in OCaml Implementations

The common OCaml data type chosen for representing matrices is `float array array`, which is the type intended by the OCaml Standard Library, which also provides the function `Array.make_matrix` to facilitate the creation and initialization of matrices of this type. A separate module `Std_matrix` is used for our implementation, containing `Std_matrix.t` as alias for arrays of float arrays and operations for handling objects of this type. The type definition is given as follows.

```
type t = float array array
```

Common Interface

In order to normalize the benchmarks as described in points 2 and 3 on the facing page, the implementations use the object-oriented features of OCaml. A common interface for all implementations of the matrix multiplication is provided by a *parameterized virtual class.*[1]

```
class virtual ['a] abstr_mmult (size:int) (label:string) =
  object
    val _size = size
    val _label = label
    method getlabel = _label
    method virtual pre : Std_matrix.t -> Std_matrix.t -> 'a * 'a
    method virtual process : 'a -> 'a -> 'a
    method virtual post : 'a -> Std_matrix.t
  end
```

Listing 3.1: Common interface for benchmark suite.

Three virtual methods are intended to be implemented in corresponding subclasses providing the following functionality.

- Method `pre` expects as input the input matrices A and B, and transforms them into a representation to fit the matrix type `'a` needed for the matrix multiplication. Also preceding run-time optimizations are implemented here, like the generation of partially unrolled loops or the offshoring of code fragments.
- Method `process` performs the actual matrix multiplication.
- Method `post` post-processes the result of the matrix multiplication and returns the product matrix C.

[1] A virtual class in OCaml is a class which has ordinary methods as well as *virtual methods*. Methods of the latter kind consist merely of a type declaration without an implementation. As shown in Listing 3.1, these classes and the corresponding virtual methods are labeled with the keyword `virtual`. Virtual classes are used to provide a common interface for methods implemented in derived subclasses. Note that in other object-oriented languages there are different terms for this type of class. A virtual class in OCaml corresponds to an *abstract class* in JAVA or a *class with pure virtual methods* in C++. Note that *virtual inheritance* in C++ refers to a special kind of multiple inheritance and does not correspond to the concept of virtual classes in OCaml (Lippman and Lajoie, 1998, Sect. 18.5).

Providing this abstract class with a type parameter `'a` allows the flexible usage of an implementation-specific matrix representation. Therefore, a subclass can be endowed with a type instantiation to set `'a` to the data type needed. As a positive side effect, this type inheritance completely eliminates polymorphism in the subclass, which enables the compiler to do optimizations for monomorphic methods. The most important optimizations in this context are the inlining of array accesses and the unboxing of float values because the type and layout of the array is known at compile time.

Inheritance Hierarchy Overview

Figure 3.1 on the next page gives an illustrating extract of the inheritance hierarchy, in form of a UML style class diagram (Open Management Group, Chap. 7). Classes are represented by boxes with three fields: the top field denotes the class name, the middle field contains the member variables and the bottom field contains the methods. The additional keyword `{virtual}` denotes a virtual class. The respective virtual methods are set in *italic* font.

Normal inheritance is represented by a solid line starting at a triangle at the superclass and leading to the corresponding subclass. For generic classes, the respective type parameter `'a` is noted in a dashed box at the upper right corner of the class box. The combined type instantiation and sub-typing uses a dashed line instead of a solid one and marks it with the stereotype `<<bind>>` (Open Management Group, Sec. 17.5.7). The type instantiation of parmeter `'a` with argument `t` is noted as `<'a -> t>`.

Unfortunately, UML lacks a suitable notation for higher-order function types because it separates parameter type annotations from the type annotation of the return value. In the figure, this deficiency was solved by giving OCaml higher-order type expressions as “return type” of methods.

Matrix Size Restriction

The restriction of matrix shapes to squares whose width is a power of 2 was chosen with the intention to simplify the implementation and presentation, especially for the recursive matrix multiplications. For the recursive algorithms, the relaxation to matrices of arbitrary shape would necessitate a padding technique to fill each matrix with additional zero lines and rows. As Chatterjee et al. (2002, Sect. 4) showed, an adaptive tiling technique can be used to minimize padding and, thus, reduce the negative effect on the execution time.

In all code examples shown below, the size of a matrix in a multiplication is determined by variable `n`, where the width is given by 2^{n}. Variable `width` is used to bind this value for subsequent usage.

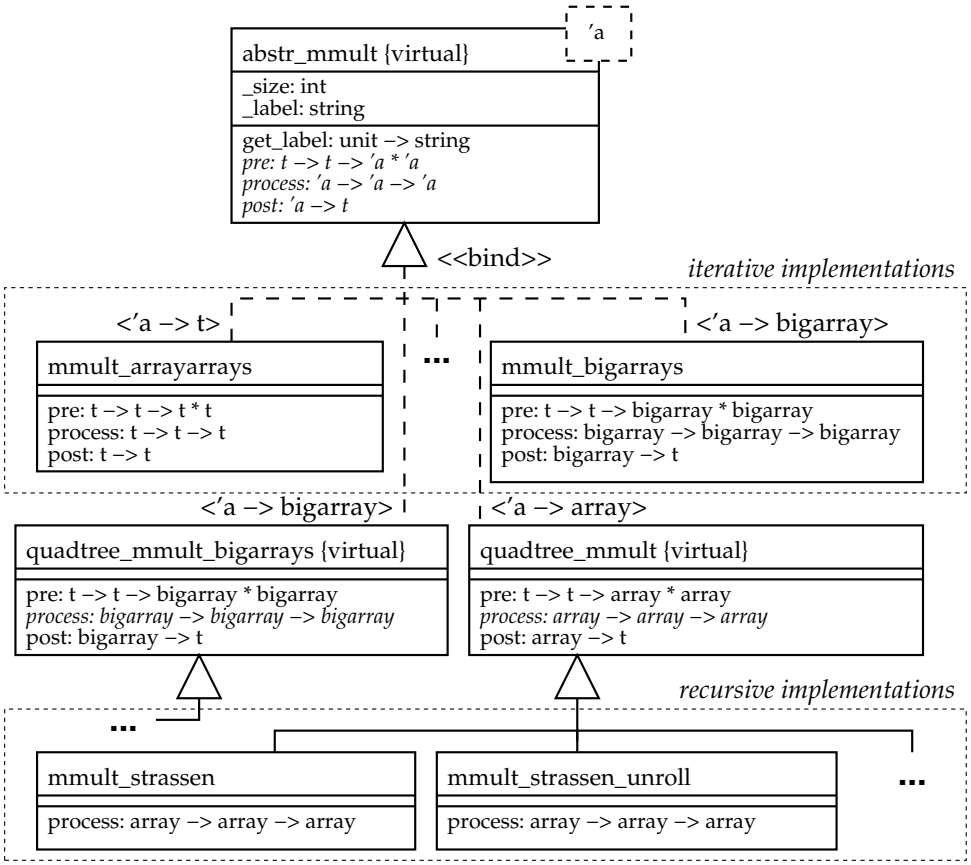


Figure 3.1: UML style class diagram extract of matrix multiplication implementations. Type names are abbreviated: *t* denotes `Std_matrix.t`, *array* denotes `float array` and *bigarray* denotes `(float, float64_elt, c_layout) Array1.t` of the `Bigarray` module.

Float arrays of arrays	
`IterOrd OAa`	naïve implementation in OCaml
`IterOrd CAa`	naïve implementation in C
Row-major order arrays	
`IterRow OA`	in OCaml using OCaml Arrays
`IterRow OB1`	in OCaml using 1-D bigarrays
`IterRow CA`	in pure C
2-dimensional arrays	
`Iter2D OB2`	in OCaml using 2-D bigarrays
`Iter2D OB2+C`	in OCaml with calculation in external C
Transposed operand matrix *B*	
`IterTrp OAa`	in OCaml using arrays of arrays
`IterTrp OB2`	in OCaml using 2-D bigarrays
`IterTrp CAa`	in C using arrays of arrays
`IterTrp CA`	in C using ordinary arrays
`IterTrp OAa+U`	in MetaOCaml using arrays of arrays and unrolling
`IterTrp OB2+C`	in OCaml with calculations in external C
`IterTrp OAa+O`	in MetaOCaml* using arrays of arrays and offshoring

* The offshoring implementation uses objects of the MetaOCaml code type.

Table 3.1: Iterative implementations of the matrix multiplication. A unique identifier is assigned to each implementation. It is shaped *impl*␣*lang type* [+ *variant*], where *impl* denotes the basic implementation, *lang* is either `O`, for (Meta)OCaml, or `C`, for C, *type* denotes the data type used for matrices and *variant* distinguishes between implementation variants.

3.2 Low-level Optimizations of the Iterative Algorithm

This section presents implementations of the matrix multiplication which are directly based on the declarative Definition 3.1 by implementing the summations by nested loops. In order to distinguish them in the subsequent benchmarks, we assign identifiers to each implementation, as shown in Table 3.1.

3.2.1 Using Arrays of Arrays

The following implementation is the first one of the benchmark suite. It directly uses the common data representation, i.e., arrays of float arrays.

```
1   method process a b =
      begin
        let c = Array.make_matrix width width 0.0 in
        for i = 0 to width-1 do
          let c_row = c.(i) in
6         let a_row = a.(i) in
          for j = 0 to width-1 do
            for k = 0 to width-1 do
```

```
            c_row.(j) <- c_row.(j) +. a_row.(k) *. b.(k).(j)
          done
        done
      done;
      c
    end
```

The implementation is closely geared to the definition of the matrix product given in Definition 3.1. Since input and output of this operation are of the type chosen as default matrix representation, no further pre- or post-processing has to be done. Though it is a direct implementation of the definition, this implementation considers a number of the speed issues discussed before: there are no polymorphic types, the rows of the matrix are of type `float array` and, therefore, are stored in a consecutive memory block of double precision floating point values. Furthermore, OCaml is aware of the special form of the loop body, i.e., despite the usual representation of float values, the intermediate results of the right hand side of the assignment will not become boxed. Also, the two-dimensional subscript of matrix `a` and `b` is split into a less frequent reference to rows in the outermost loop and a more frequent reference to column entries in the innermost loop. Unfortunately, the row subscript of matrix `b` cannot easily be moved to an outer loop level because it is dependent on the inner loop index `k`.

Using arrays of arrays, the computation structure can be written in C very much like the OCaml implementation.

```
for (i = 0; i < width; i++) {
  double * c_row = c[i];
  double * a_row = a[i];
  for (j = 0; j < width; j++) {
    c_row[j] = 0;
    for (k = 0; k < width; k++) {
      c_row[j] += a_row[k] * b[k][j];
    }
  }
}
```

Aside from the syntax, the most obvious differences occur in the initialization of array `c` with 0 within the loop of `j` and the special nature of C arrays as pointers to consecutive chunks of memory, indicated by the data type `double *` (i.e., pointer to `double`). Therefore, an array of arrays in C is actually an array of pointers with each pointer addressing a one-dimensional array of matrix row entries.

Using arrays of arrays as matrix representation is not as easy in C as it is in OCaml. Because C does not provide automated memory management like the OCaml garbage collection, matrices of dynamic size have to be allocated and freed manually. This task is complex and error-prone, especially for data structures with many pointers. Therefore, preceding the code fragment above, a number of consecutive memory segments has to be allocated, not only for the array of pointers but also for each matrix row.

Arrays of arrays provide a way of indexing that is quite similar to the subscript of a matrix used in mathematical notation. Still, operations on them need special care by the implementer due to the risk of irregular shape and aliasing. Because either rows or columns are represented by individual arrays, an irregular shape occurs if two or more of these arrays have different lengths. The matrix represented by the array of

arrays would not be rectangular anymore. By using the function `Array.make_matrix` a rectangular shape can be guaranteed, but nothing can be said about the layout of rows as chunks in the memory. Notably, these chunks need not occupy a consecutive span of memory. Aliasing is another issue since arrays are mutable objects. Array elements with different indices `i` and `j` may be of the same value and, therefore, in an array of arrays, `a.(i)` can refer to the same array as `a.(j)`. Any modification of the first array would affect the latter.

The next subsection uses representations which prevent the risk of both irregular shape and aliasing.

3.2.2 Using One- and Multi-dimensional Arrays

Languages like C and Fortran provide "real" multi-dimensional arrays. In the case of C they have to be allocated statically (i.e., on the stack) and their shape must be known statically. An array of this kind is arranged as a consecutive chunk of memory, which is indexed in row-major order (in C) or column-major order (in Fortran). In OCaml, the built-in arrays are restricted to one dimension. In addition, the bigarray library offers arrays of arbitrary length and dimensions from 1 to 16, laid out either in row-major or column-major order.

In the case that a language does not provide multi-dimensional arrays, a simple way of representing a multi-dimensional matrix by a one-dimensional array is to use a mapping from two-dimensional matrix indices to the one-dimensional indices of the array. The only prerequisite is that it corresponds to following definition.

3.2 Definition (Index Function)
Function $f \in \overline{n} \times \overline{m} \to \mathbb{N}$ is called *index function* of array $a \in \mathbf{R}^{\overline{k}}$ if both

$$f \text{ is injective}$$

and

$$\{f(p,q) \mid (p,q) \in \overline{n} \times \overline{m}\} \subseteq \overline{k},$$

i.e., the image of f is a subset of range $\overline{k}$.

□

In order to save memory, the cardinality of the index space of the two-dimensional matrix should be equal to the length of the array, i.e., $nm = k$.

Now we can define a 2D matrix representation by an array and an index function as follows.

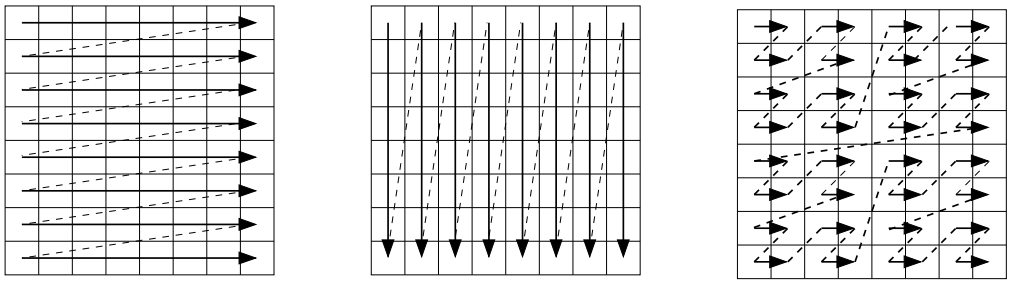


Figure 3.2: Memory layout of matrices for different index functions

3.3 Definition (2D Matrix Representation)
Consider a matrix $M \in \mathbf{R}^{\overline{v} \times \overline{w}}$ and an index function $f \in \mathbb{N}^{\overline{v} \times \overline{w}}$ of an array $a \in \mathbf{R}^{\overline{v} \cdot \overline{w}}$. Then a and f are a *2D matrix representation* of M if and only if

$$a_{f(i,j)} = M_{i,j} \qquad \forall i \in \overline{v}, j \in \overline{w}$$

□

A one-dimensional array with a two-dimensional index function not only provides a means of compensation for the absence of an appropriate language construct. In addition, there is no restriction to a specific layout of the array. In order to lay out the matrix in row-major order, the index function to be used would be $f(i,j) = i \cdot w + j$. This function allows to index matrices of width w and an arbitrary number of rows. A column-major indexing for a matrix of height h would be $f(i,j) = i + j \cdot h$. Another layout and index function is the z-Morton index used for recursive algorithms (see Figure 3.2).

Row-major Order Matrix Multiplication

A matrix multiplication using one-dimensional OCaml arrays is specified by the following code fragment.

```
  method process a b =
    begin
      let c = Array.make ( width * width ) 0.0 in
      for i = 0 to width - 1 do
        let i_of = width * i in
        for j = 0 to width - 1 do
          for k = 0 to width - 1 do
            c.(i_of + j) <- c.(i_of + j)
              +. a.(i_of + k) *. b.(k * width + j)
          done;
        done;
      done;
      c
    end
```

This implementation uses the row-major order index function. For optimization, the index function was not only inlined but also optimized by moving the calculation of the offset `i_of` of row `i` to the outermost loop.

Unfortunately, the standard arrays provided in OCaml are restricted to a maximum length which is strongly dependent on the target architecture.[2] This limit is much more eminent for one-dimensional arrays than for arrays of arrays. On a 32-bit i386 machine, the limit is exceeded by an array of length 2^{22}. Thus, a matrix with index space $\overline{2^{11}} \times \overline{2^{11}}$ can not be represented by a one-dimensional array. If we keep the restriction of the width being a power of 2, arrays of arrays can keep matrices of any size up to $2^{21} \cdot 2^{21} = 2^{42}$.

Two-dimensional Matrix Multiplication

OCaml bigarrays are not subject to this restriction, and the corresponding code fragment can be adopted easily using bigarrays by writing the subscript operator of OCaml arrays with the braces of the respective bigarray operator.

```
method process a b =
  begin
    let c =
      Array1.create float64 c_layout (width * width)
    in
    for i = 0 to width-1 do
      let i_of = i * width in
      for j = 0 to width-1 do
        let ij_of = i_of + j in
        c.{ij_of} <- 0.0;
        for k = 0 to width-1 do
          c.{ij_of} <-
            c.{ij_of} +. a.{i_of + k} *. b.{k * width + j}
        done
      done;
    done;
    c
  end
```

Again, using either standard arrays or bigarrays, the respective array type should be monomorphic to enable OCaml to inline operations on them. While the standard arrays are of type `float array`, the corresponding bigarrays are of type `(float, float64_elt,c_layout) Array1.t`. This is assured by the use of the respective `create` function and by explicit type constraints on the parameters of the enclosing function.

For comparison, a matrix multiplication in C using row-major order index function, is given by the following loop program.

[2] The maximum array length can be requested by the variable `Sys.max_array_length`.

```
    for (i = 0; i < width; i++) {
2     i_of = width * i;
      for (j = 0; j < width; j++) {
        int ij_of = i_of + j;
        c[ij_of] = 0;
        for (k = 0; k < width; k++) {
7         c[ij_of] += a[i_of + k] * b[k * width + j];
        }
      }
    }
```

Though C provides multi-dimensional arrays, the use of one-dimensional arrays in C is very common due to the following reasons:

- Many compilers expect the dimensions of static arrays to be given as constant literals. Also, two-dimensional arrays as function arguments must have at least a constant width, so the row-major indexing is statically known at compile time.
- The access of a one-dimensional array by a user-defined index function is more flexible because it can be chosen according to the needs of the program.
- The dynamic allocation of one-dimensional arrays is simpler than imitating two-dimensional arrays with a complex data structure like an array of arrays.

The C99 standard (ISO/IEC JTC1/SC22/WG14, 1999) introduces variable-sized arrays, which could motivate a more frequent use of real multi-dimensional arrays in C in the future. This feature also allows the passing of variable-sized multi-dimensional arrays as function argument, a big advantage for writing reusable code. Unfortunately, this feature is still not implemented in all popular C compilers.

Two-dimensional arrays as a language construct help the implementer of numerical problems to program the mathematics involved more directly. Another advantage over other representations is the awareness of the compiler, which may perform optimizations based on the knowledge about a specific matrix shape. Still, a naïve implementation of the matrix multiplication using bigarrays and exclusively two-dimensional indexing, did not prove to run as efficient as the following version.

```
    for i = 0 to width-1 do
      let c_row = Array2.slice_left c i in
      let a_row = Array2.slice_left a i in
      for j = 0 to width-1 do
5       for k = 0 to width-1 do
          c_row.{j} <- c_row.{j} +. a_row.{k} *. b.{k, j}
        done
      done
    done;
```

This nest of loops selects rows at an outer loop level, just like the example with arrays of arrays. In this case, we make use of the function `Array2.slice_left`, which provides a one-dimensional view of a single matrix row.

Exporting OCaml Bigarrays to C

Bigarrays not only provide multi-dimensionality and huge array sizes. In fact, the main purpose of bigarrays is to simplify the interfacing to C programs, because they can easily be created to fit the needs of the C function to be called from OCaml. The interfacing mechanism provided by OCaml is described in detail later on (Section 6.2).

For matrix multiplication, it is sensible to do the pre- and post-processing within OCaml, i.e., to convert the input matrices to bigarrays and the result of the product back to OCaml arrays of arrays. The actual time-consuming multiplication can be done in C and made known to OCaml by the following declaration.

```
type flatm = (float, float64_elt, c_layout) Array2.t
external mmult :
  int -> flatm -> flatm -> flatm = "mmult_stub"
```

Because we want to use them on the C side, the internal representation of bigarrays is chosen accordingly. The concrete type argument `float64_elt` defines the array elements to be of C type `double`. The type argument `c_layout` sets the layout of the matrix to be ordered row major.

A C stub function implements the extractions of C arrays from the respective bigarray argument and the initialization of the result matrix. In this example, the actual row-major matrix multiplication is embedded in the stub code at lines 15–23.

```
  value mmult_stub (value n, value big_a, value big_b) {
2   int i, j, k, i_of;
    int width = 1 << Int_val(n);
    long dims[] = {width,width};

    value big_c = alloc_bigarray(
7       BIGARRAY_FLOAT64|BIGARRAY_C_LAYOUT,
        2, NULL, dims);

    double* a = (double*) Bigarray_val(big_a)->data;
    double* b = (double*) Bigarray_val(big_b)->data;
12  double* c = (double*) Bigarray_val(big_c)->data;
    CAMLparam3(n, big_a, big_b);

    for (i = 0; i < width ; i++) {
      i_of = width * i;
17    for (j = 0; j < width ; j++) {
        int ij_of = i_of + j;
        c[ij_of] = 0;
        for (k = 0; k < width ; k++) {
          c[ij_of] += a[i_of + k] * b[k * width + j];
22      }
      }
    }
    CAMLreturn(big_c);
  }
```

3.2.3 Using Transposed Matrices

As shown in the previous examples, the number of operations can be reduced significantly by moving either the extraction of rows or the calculation of row offsets to an

outer loop. Unfortunately, this is not possible for the subscript of the right operand matrix B because the iteration over the rows is done by the innermost loop index. Furthermore, this non-local access is especially bad in terms of data-cache misses.

On the other hand, the column index of matrix B is incremented less frequently, being the loop index of the middle loop. So, a representation of B by a column-major matrix would not only allow the extraction of offset calculations but would also be better in terms of locality for memory accesses. Because a matrix representation in column-major order corresponds to a transposed matrix in row-major order, B has to be transposed in the pre-processing phase in order to allow this kind of optimization.

```
      method pre a b =
        let tb =
          Array.init width begin fun i ->
4           Array.init width begin fun j ->
              b.(j).(i)
            end
          end in
        (a,tb)
```

This preprocessing method combines the creation of a new array of arrays matrix with the transposition of matrix b. Note that this pre-processing method did not undergo any extra optimizations, due to the fact that the cubic number of operations of the matrix multiplication dominates the quadratic number of operations of this transposition.

However, the matrix multiplication with a column-major B should be optimized accordingly.

```
  method process a b =
2   begin
      let width = 1 lsl size in
      for i = 0 to width - 1 do
        let c_row = c.(i) in
        let a_row = a.(i) in
7       for j = 0 to width - 1 do
          let b_col = b.(j) in
          for k = 0 to width - 1 do
            c_row.(j) <- c_row.(j) +. a_row.(k) *. b_col.(k)
          done
12      done
      done;
      c
    end
```

Modeled on the previous examples, the benchmark suite also contains a transposed matrix multiplication for two-dimensional bigarrays, two-dimensional bigarrays with an external multiplication in C and a pure C implementation.

3.3 High-level Optimizations by Recursive Algorithms

3.3.1 Ordinary and Strassen's Matrix Multiplication

Another way of calculating the matrix product $C = A \cdot B$ is to use an algorithm which is based on a recursive definition. A non-trivial matrix multiplication is defined recur-

sively in terms of the four quadrants of each participating matrix.

3.4 Definition (Ordinary Recursive Matrix Multiplication)
For any matrix M and $i, j \in \{0, 1\}$, let $M^{i,j}$ denote a quadrant of matrix M, i.e.,

$$\begin{pmatrix} M^{0,0} & M^{0,1} \\ M^{1,0} & M^{1,1} \end{pmatrix} = M.$$

Let $A \in \mathbf{R}^{\overline{u} \times \overline{v}}$ and $B \in \mathbf{R}^{\overline{v} \times \overline{w}}$. The matrix multiplication $C = A \cdot B$ is defined by the following equations

$$\begin{aligned} C^{0,0} &= A^{0,0} \cdot B^{0,0} + A^{0,1} \cdot B^{1,0} \\ C^{0,1} &= A^{0,0} \cdot B^{0,1} + A^{0,1} \cdot B^{1,1} \\ C^{1,0} &= A^{1,0} \cdot B^{0,0} + A^{1,1} \cdot B^{1,0} \\ C^{1,1} &= A^{1,0} \cdot B^{0,1} + A^{1,1} \cdot B^{1,1} \end{aligned}$$

In order to make this definition sound, the shapes of all matrices must fit according to

$$\begin{aligned} w(A^{0,0}) = w(A^{1,0}) = h(B^{0,0}) = h(B^{0,1}) \\ w(A^{0,1}) = w(A^{1,1}) = h(B^{1,0}) = h(B^{1,1}). \end{aligned}$$

Furthermore, the multiplication of matrices of size 1 corresponds to a multiplication of both respective entries in $\mathbf{R}$. Any multiplication or summation of matrices with zero width or height, yields, again, an empty matrix.

□

So, the recursive equations reduce the problem of multiplying the whole matrices to a number of multiplications and summations of sub-matrices. The following Lemma states that this definition of a matrix multiplication is equivalent to the iterative one given above.

3.5 Lemma
The equations given in Definition 3.4 hold for the iterative matrix multiplication of Definition 3.1 on page 31.
Idea of Proof:
Let $v = w(A) = h(B)$ and $v' = w(A^{0,0}) = h(B^{0,0})$. For all $i \in \overline{h(A^{0,0})}$ and $j \in \overline{w(B^{0,0})}$

$$\begin{aligned} & (A^{0,0} \cdot B^{0,0})_{i,j} + (A^{0,1} \cdot B^{1,0})_{i,j} \\ &= \sum_{k=0}^{v'-1} A^{0,0}_{i,k} \cdot B^{0,0}_{k,j} + \sum_{k=0}^{v-v'-1} A^{0,1}_{i,k} \cdot B^{1,0}_{k,j} && \text{Def. 3.1} \\ &= \sum_{k=0}^{v'-1} A_{i,k} \cdot B_{k,j} + \sum_{k=v'}^{v-1} A_{i,k} \cdot B_{k,j} && \text{sub-matrix definition} \\ &= \sum_{k=0}^{v-1} A_{i,k} \cdot B_{k,j} = C_{i,j} = C^{0,0}_{i,j} && \text{Def. 3.1} \end{aligned}$$

The proofs for the equations of $C^{0,1}$, $C^{1,0}$ and $C^{1,1}$ are similar and are omitted here.

□

A direct implementation of the equations in Definition 3.4 on the facing page by a recursive function would have 8 recursive calls and 4 summations in the recursive case. The restriction on the matrix shape and size, given in the design decisions on page 32, simplifies the implementation: a square matrix whose width is a power of 2 can be sectioned by cutting it into four equally sized squares. The termination case can then be implemented as a single floating-point multiplication of scalar values. For this algorithm, the total number of operations $T(N)$ for square matrices of width N is given by the following equations:

$$\begin{aligned} T(1) &= c_1 \\ \forall\, N > 1: \quad T(N) &= \underbrace{8 \cdot T\left(\frac{N}{2}\right)}_{\substack{\text{recursive} \\ \text{multiplications}}} + \underbrace{4 \cdot c_2 \cdot N^2}_{\substack{\text{matrix} \\ \text{summations}}} \end{aligned}$$

where $c_1, c_2 > 0$ denote a constant number of operations.

To obtain a rough estimate of the number of operations in a closed form, we can use the following special case of the *Master Theorem* (Cormen et al., 2001, Sec. 4.3).

3.6 Theorem
Let $\alpha \geq 1$ and $\beta > 1$. Let $f : \mathbb{N} \to \mathbb{R}$ be a non-negative function with $f(N) \in O(N^{\log_\beta \alpha - \epsilon})$ for a suitable $\epsilon > 0$. If $T : \mathbb{N} \to \mathbb{R}$ is a function which satisfies the recursive equation

$$T(N) = \alpha \cdot T\left(\frac{N}{\beta}\right) + f(N)\,,$$

then $T(N)$ can be determined asymptotically as

$$T(N) \in \Theta(N^{\log_\beta \alpha}).$$

□

According to this theorem, for $\alpha = 8$, $\beta = 2$ and $\epsilon = 1$, the number of operations of the recursive matrix multiplication is in $\Theta(N^{\log_2 8}) = \Theta(N^3)$. Note that this algorithm belongs to the same complexity class as the iterative matrix multiplication.

Strassen's version of the matrix multiplication is also defined recursively, according to the following definition (Press et al., 1989, Sect. 2.11).

3.7 Definition (Strassen's Matrix Multiplication)
Given the matrix decomposition of A, B, C according to Definition 3.4, a matrix multiplication $C = A \cdot B$ is defined by the following equations:

$$
\begin{aligned}
M^0 &= (A^{0,0} + A^{1,1}) \cdot (B^{0,0} + B^{1,1}) \\
M^1 &= (A^{1,0} + A^{1,1}) \cdot B^{0,0} \\
M^2 &= A^{0,0} \cdot (B^{0,1} - B^{1,1}) \\
M^3 &= A^{1,1} \cdot (B^{1,0} - B^{0,0}) \\
M^4 &= (A^{0,0} + A^{0,1}) \cdot B^{1,1} \\
M^5 &= (A^{1,0} - A^{0,0}) \cdot (B^{0,0} + B^{0,1}) \\
M^6 &= (A^{0,1} - A^{1,1}) \cdot (B^{1,0} + B^{1,1}) \\
C^{0,0} &= M^0 + M^3 - M^4 + M^6 \\
C^{0,1} &= M^2 + M^4 \\
C^{1,0} &= M^1 + M^3 \\
C^{1,1} &= M^0 + M^2 - M^1 + M^5
\end{aligned}
$$

□

This definition is equivalent to the ordinary recursive matrix multiplication, which can be shown by inserting M^0 to M^6 into the last four equations. The advantage over the ordinary multiplication is a reduction of the total number of operations. Though Strassen uses more additions and subtractions (which are all $O(N^2)$), the number of recursive calls is reduced to 7. According to the Master Theorem, this yields a number of operations which is in $\Theta(N^{\log_2 7})$, where $\log_2 7 \approx 2.81$.

Unfortunately, the memory consumption of Strassen's matrix multiplication is higher than for the ordinary recursive algorithm because memory has to be allocated for intermediate results, for each recursive step.

3.3.2 Data Structures for Recursive Matrix Multiplications

Another advantage of recursive matrix-multiplication is the chance of increasing the locality of memory accesses. To achieve this, using a matrix representation which is row or column oriented would be inappropriate. The strategy of solving the multiplication recursively by reducing the problem to quarters of matrices is neither row- nor column-oriented (Figure 3.3 on page 49).

An adequate data structure for a matrix M is a so called *quad-tree*. This tree has a single root, representing M as a whole. Each inner node has four children, the subsections $M^{0,0}$, $M^{0,1}$, $M^{1,0}$ and $M^{1,1}$ of M. The leaves, finally, consist of M's element entries.

For the actual implementation this abstract concept of a quad-tree was not implemented by a recursive OCaml type definition. Instead, one-dimensional arrays indexed by a *z-Morton index function* were used (Chatterjee et al., 2002). As the following theorem shows, this representation resembles quad-trees. In addition, the acquisition of

child nodes in a quad-tree is equivalent to partitioning the array into four sub-arrays of equal length.

3.8 Definition (z-Morton)
Let $\mathcal{B}_n(i) \in \{0,1\}^{\overline{n}}$ denote the bit representation of i, let $\mathcal{B}^{-1}(b)$ denote the binary interpretation of an arbitrary sized bitmap b and let $a \bowtie b$ be the bit-wise interleaving of the bitmaps $a, b \in \{0,1\}^{\overline{n}}$.[3]
Then, the *z-Morton function* $z_n : \overline{2^n} \times \overline{2^n} \to \overline{2^{2 \cdot n}}$ is defined by

$$z_n(i,j) = \mathcal{B}^{-1}(\mathcal{B}_n(i) \bowtie \mathcal{B}_n(j)).$$

$z_n(i,j)$ is called *z-address* of coordinates (i,j).

□

Figure 3.4 on page 50 sketches this computation of a z-address. To give an example for actual values, consider Figure 3.3 on page 49. For the coordinate $(2,6)$, we calculate the z-Morton index by interleaving the three-bit representations of 2 and 6.

$$z_3(2,6) = \mathcal{B}^{-1}(\mathcal{B}_3(2) \bowtie \mathcal{B}_3(6)) = \mathcal{B}^{-1}(\langle 010 \rangle \bowtie \langle 110 \rangle) = \mathcal{B}^{-1}(\langle 011100 \rangle) = 28$$

The use of an array with a z-Morton index for representing a matrix has a number of characteristics, which are important for the implementation of a recursive matrix multiplication.

3.9 Theorem
Let array $a \in \mathbf{R}^{\overline{2^{2 \cdot n}}}$ and z-Morton function z_n be a 2D matrix representation of $M \in \mathbf{R}^{\overline{2^n} \times \overline{2^n}}$. Then the following propositions are true.

1. z_n is bijective. *Proof:* bijectivity of $\mathcal{B}_n$ and $\bowtie$.

2. For $n > 0$, M can be split recursively into four quadrants $M^{0,0}$, $M^{0,1}$, $M^{1,0}$ and $M^{1,1}$ of size $\overline{2^{n-1}} \times \overline{2^{n-1}}$, where

$$\begin{pmatrix} M^{0,0} & M^{0,1} \\ M^{1,0} & M^{1,1} \end{pmatrix} = M$$

   where

$$M^{k,l}_{i,j} := M_{i+k \cdot 2^{n-1}, j+l \cdot 2^{n-1}} \qquad \forall k,l \in \{0,1\} \text{ and } i,j \in \overline{2^{n-1}}$$

   Proof: We express the mapping of the index of a sub-matrix to the index of M by function

$$f_{k,l}(i,j) := (i + k \cdot 2^{n-1}, j + l \cdot 2^{n-1}).$$

[3] This corresponds to the notation defined in Section 1.1. Note that using $\mathcal{B}^{-1}$ instead of the real inverse of $\mathcal{B}_{2n}$ is motivated by the fact that $\mathcal{B}^{-1}_{2n}(b) = \mathcal{B}^{-1}(b)$ for all $b \in \{0,1\}^{\overline{2 \cdot n}}$.

We have to show that the images $f_{k,l}(\overline{2^{n-1}} \times \overline{2^{n-1}})$, for all $k, l \in \{0,1\}$, describe a partition of $\overline{2^n} \times \overline{2^n}$, which is the index space of M. This can be achieved by evaluating each $f_{k,l}$ with the respective range bounds:

$$\begin{aligned}
f_{0,0}(\overline{2^{n-1}} \times \overline{2^{n-1}}) &= \{0, \ldots, 2^{n-1} - 1\} \times \{0, \ldots, 2^{n-1} - 1\} \\
f_{0,1}(\overline{2^{n-1}} \times \overline{2^{n-1}}) &= \{0, \ldots, 2^{n-1} - 1\} \times \{2^{n-1}, \ldots, 2^n - 1\} \\
f_{1,0}(\overline{2^{n-1}} \times \overline{2^{n-1}}) &= \{2^{n-1}, \ldots, 2^n - 1\} \times \{0, \ldots, 2^{n-1} - 1\} \\
f_{1,1}(\overline{2^{n-1}} \times \overline{2^{n-1}}) &= \{2^{n-1}, \ldots, 2^n - 1\} \times \{2^{n-1}, \ldots, 2^n - 1\}
\end{aligned}$$

3. Let arrays $a^{0,0}$, $a^{0,1}$, $a^{1,0}$ and $a^{1,1}$ be obtained by splitting a in four equally sized sub-arrays, i.e.,

$$a^{k,l}_i := a_{(2 \cdot k + l) \cdot 2^{n-2} + i} \qquad \forall k, l \in \{0,1\}, i \in \overline{2^{2 \cdot n - 2}}.$$

Then $a^{k,l}$ and z_{n-1} are a 2D matrix representation of $M^{k,l}$ for all $k, l \in \{0,1\}$.
Proof:

$$\begin{aligned}
M^{k,l}_{i,j} &= M_{i+k \cdot 2^{n-1}, j+l \cdot 2^{n-1}} && \text{definition of } M^{k,l} \\
&= a_{z_n(i+k \cdot 2^{n-1}, j+l \cdot 2^{n-1})} && a \text{ and } z_n \text{ represent } M \\
&= a_{\mathcal{B}^{-1}(\mathcal{B}_n(i+k \cdot 2^{n-1}) \bowtie \mathcal{B}_n(j+l \cdot 2^{n-1}))} && \text{Def. 3.8} \\
&= a_{\mathcal{B}^{-1}(\langle k \rangle \hat{} \mathcal{B}_{n-1}(i) \bowtie \langle l \rangle \hat{} \mathcal{B}_{n-1}(j))} && \text{extraction of most significant digits} \\
&= a_{\mathcal{B}^{-1}(\langle kl \rangle \hat{} (\mathcal{B}_{n-1}(i) \bowtie \mathcal{B}_{n-1}(j)))} && \langle k \rangle \hat{} a \bowtie \langle l \rangle \hat{} b = \langle kl \rangle \hat{} (a \bowtie b) \\
&= a_{(2 \cdot k + l) \cdot 2^{n-2} + \mathcal{B}^{-1}(\mathcal{B}_{n-1}(i) \bowtie \mathcal{B}_{n-1}(j))} && \mathcal{B}^{-1}(a \hat{} b) = \mathcal{B}^{-1}(a) \cdot 2^{\#b} + \mathcal{B}^{-1}(b) \\
&= a^{k,l}_{\mathcal{B}^{-1}(\mathcal{B}_{n-1}(i) \bowtie \mathcal{B}_{n-1}(j))} && \text{definition of } a^{k,l} \\
&= a^{k,l}_{z_{n-1}(i,j)} && \text{Def. 3.8}
\end{aligned}$$

4. Matrix M forms a quad-tree in the following sense. Each node corresponds to a sub-matrix. A leaf corresponds to a matrix with only one entry. The children of an inner node N are the corresponding sub-matrices $N^{0,0}$, $N^{0,1}$, $N^{1,0}$ and $N^{1,1}$.

5. Let $i, j \in \overline{2^n}$, which have the bit representation

$$\begin{aligned}
\mathcal{B}_n(i) &= \langle i_{n-1} \ldots i_0 \rangle \\
\mathcal{B}_n(j) &= \langle j_{n-1} \ldots j_0 \rangle.
\end{aligned}$$

Also, let

$$M^{\langle i_{n-1} j_{n-1} \ldots i_0 j_0 \rangle} := \left(\ldots \left(\left(M^{i_{n-1}, j_{n-1}} \right)^{i_{n-2}, j_{n-2}} \right) \ldots \right)^{i_0, j_0}$$

be a leaf in the quad-tree of M. Then the following equation is true:

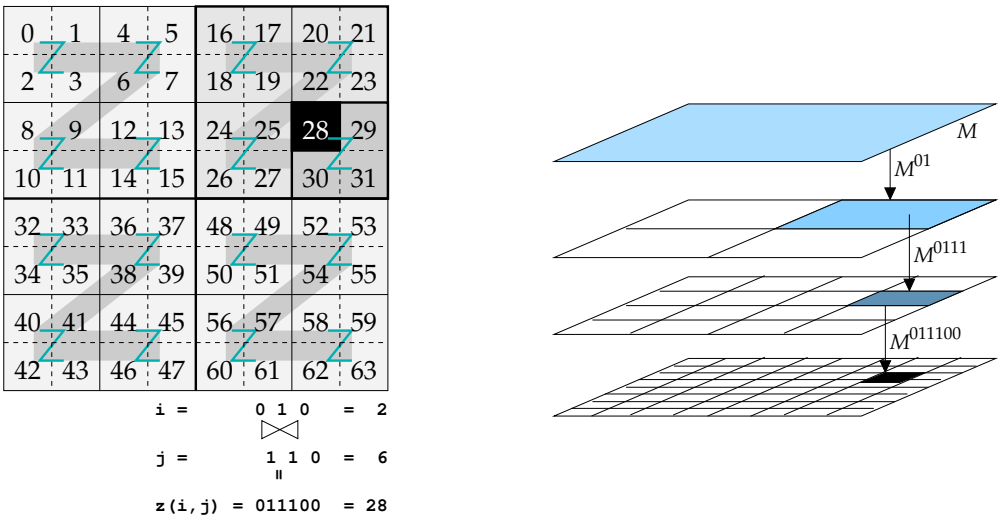


Figure 3.3: z-Morton index and quad-tree analogy

$$M_{0,0}^{\mathcal{B}_{2\cdot n}(z_n(i,j))} = M_{i,j}$$

In other words, the bit-representation of the z-Morton index of (i,j) is the quad-tree path to $M_{i,j}$.

Proof by induction on n: For the base case $n = 1$, we can use point 2 of this theorem:

$$M_{0,0}^{\langle i_0 j_0\rangle} = M_{0,0}^{i_0,j_0} = M_{0+i_0\cdot 2^{1-1},0+j_0\cdot 2^{1-1}} = M_{i_0,j_0}$$

As hypothesis for the inductive case, the proposition is true for all $i,j \in \overline{2^{n'}}$ with $n' < n$. Then,

$$\begin{aligned}
& M_{0,0}^{\langle i_{n-1}j_{n-1}i_{n-2}j_{n-2}\ldots i_0 j_0\rangle} \\
&= (M^{i_{n-1},j_{n-1}})_{0,0}^{\langle i_{n-2}j_{n-2}\ldots i_0 j_0\rangle} \\
&= M^{i_{n-1},j_{n-1}}_{\mathcal{B}^{-1}\langle i_{n-2}\ldots i_0\rangle,\mathcal{B}^{-1}\langle j_{n-2}\ldots j_0\rangle} && \text{induction hypothesis} \\
&= M_{\mathcal{B}^{-1}\langle i_{n-2}\ldots i_0\rangle + i_{n-1}\cdot 2^{n-1},\mathcal{B}^{-1}\langle j_{n-2}\ldots j_0\rangle + j_{n-1}\cdot 2^{n-1}} && \text{point 2 of this theorem} \\
&= M_{\mathcal{B}^{-1}\langle i_{n-1}i_{n-2}\ldots i_0\rangle,\mathcal{B}^{-1}\langle j_{n-1}j_{n-2}\ldots j_0\rangle} && \text{definition of } \mathcal{B}^{-1} \text{ in Section 1.4} \\
&= M_{i,j}
\end{aligned}$$

□

Figure 3.3 demonstrates the quad-tree analogy by showing how we can obtain the z-address of $(2,6)$ by splitting M recursively and writing down the quad-tree path. First,

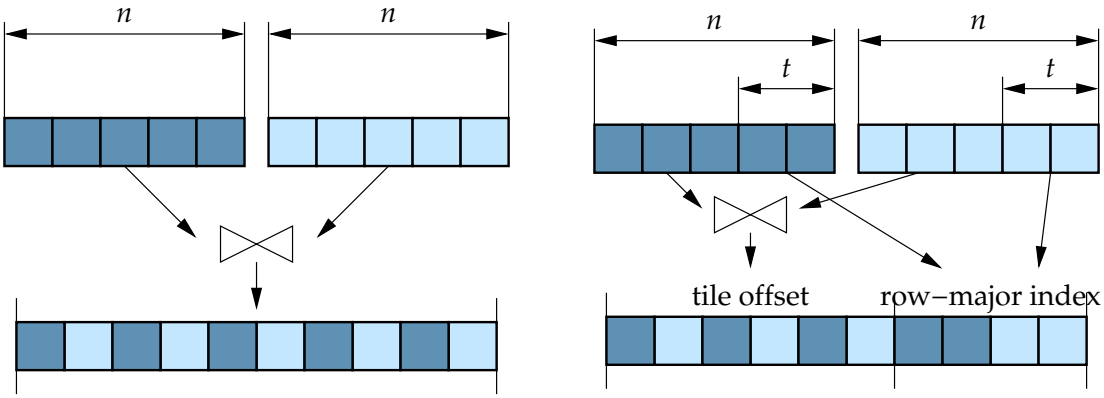


Figure 3.4: Computation of untiled and tiled z-Morton index by bit interleaving for matrix width 2^n and tile width 2^t with $n = 5$ and $t = 2$

we extract sub-matrix $M^{\langle 01 \rangle}$, then sub-matrix $M^{\langle 0111 \rangle}$ and finally sub-matrix $M^{\langle 011100 \rangle}$. According to Theorem 3.9, we get the z-address by interpreting this quad-tree path:

$$z_3(2,6) = \mathcal{B}^{-1}(\langle 011100 \rangle) = 28$$

Due to the fact that a z-Morton indexed array is an actual implementation of a quad-tree and that it is the only implementation presented in this thesis, both terms are used interchangeably in the rest of this thesis.

3.3.3 Recursive Matrix Multiplication with Tiling

As each input matrix, as well as each matrix used as intermediate result, reduces to 1/4 in size for each recursive step, there is a predefined depth, at which it completely fits in the cache. At this point, it is more and more unlikely to gain further from the recursive computation and memory layout. Though we may still benefit from the reduced complexity of Strassen's algorithm, the risk of cache misses disappears, even for a row-order or column-order layout. Furthermore, as Chatterjee et al. (2002) showed, pruning the recursion well before the element level and multiplying iteratively reduces the execution time distinctively. The main reason for this effect is the absence of the recursion overhead. A linearization of the cascading recursion is not performed by the OCaml compiler, so the run-time stack still has to be maintained. For the Strassen implementation, there is additional overhead in the allocation of temporary memory for intermediate results.

In order to decrease the overhead of addressing, the modified version of the z-Morton index function given by Chatterjee et al. (2002, Sec. 3.2) was used, which coincides with a row-major indexing for sub-matrices smaller than a given tile size.

3.10 Definition (Tiled z-Morton Index)
The *tiled z-Morton index function* $z_{n,t} : \overline{2^n} \times \overline{2^n} \to \overline{2^{2n}}$ for square tiles of width 2^t is defined as follows:

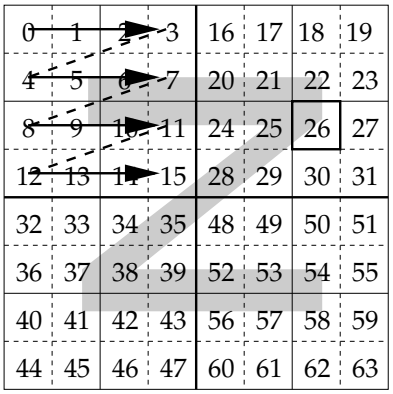


Figure 3.5: Tiled z-Morton index with tile width $2^2 = 4$.

$$z_{n,t}(i,j) = \underbrace{2^{2t} \cdot z_{n-t}(i_{\text{high}}, j_{\text{high}})}_{\text{offset of tile}} + \underbrace{2^t \cdot i_{\text{low}} + j_{\text{low}}}_{\text{row-major index}}$$

where

$$(i_{\text{high}}, i_{\text{low}}) = (i \operatorname{div} 2^t, i \bmod 2^t)$$
$$(j_{\text{high}}, j_{\text{low}}) = (j \operatorname{div} 2^t, j \bmod 2^t)$$

□

The calculation of this index function is illustrated in Figure 3.4 on the preceding page and the effect on the layout can be seen in Figure 3.5. In this example, the matrix width is $2^3 = 8$ and the tile width is $2^2 = 4$. Therefore, the recursive layout implied by $z_{3,2}$ is pruned after only one sectioning step. For each sub-matrix, the entries are laid out in row-major order, while the four matrices themselves are sequenced in z-order. Note that, for a tile width of 2^0, the tiled z-Morton function is equal to its non-tiled version, i.e.,

$$z_{n,0} = z_n \qquad \forall n \geq 0.$$

3.3.4 Implementation Issues

Table 3.2 on the next page summarizes all recursive matrix multiplications which were included in the benchmark suite. As stated before, for all of these implementations quad-trees were used, i.e., one-dimensional arrays with tiled z-Morton layout. The conversion of a two-dimensional `float array array` to a one-dimensional quad-tree array is performed by the two functions `zmorton` and `unzmorton`, which do all index calculations. These functions resemble the function $z_{n,t}$ and its inverse $z_{n,t}^{-1}$. Because

Ordinary matrix multiplication	
`RecOrd OA`	using standard arrays
`RecOrd OB1`	using 1-D bigarrays
Strassen's matrix multiplication	
`RecStr OA`	using standard arrays
`RecStr OB1`	using 1-D bigarrays
`RecStr OA+U`	using standard arrays and unrolling
`RecStr OB1+C`	using 1-D bigarrays and tile multiplication in C
`RecStr OA+O`	using standard arrays and offshoring

Table 3.2: Recursive implementations of the matrix multiplication.

they do not dominate the overall matrix multiplication, no further optimization of these functions was considered. Therefore, only the OCaml type signatures are given here.

```
type tile_layout = RowMajor | ColumnMajor
val zmorton : ?tile_n:int -> ?tile_order:tile_layout ->
  int -> int * int -> int
val unzmorton : ?tile_n:int -> ?tile_order:tile_layout ->
  int -> int -> int * int
```

For a matrix with index space $\overline{2^n} \times \overline{2^n}$, `zmorton n (i,j)` returns the z-address $z_{\mathtt{n}}(\mathtt{i},\mathtt{j})$ by interleaving the bit representations of `i` and `j`: the bits of `i` make up the odd bits of the z-address returned, the bits of `j` make up the even bits. In order to compute the z-address taking account of tiling, function `zmorton` provides the two optional parameters[4] `tile_n`, to allow the specification of a tile width, and `tile_order`, to set the order of elements within a tile to either row-major or column-major. The application of

```
zmorton ~tile_n:t ~tile_order:ord n (i,j)
```

returns the z-address $z_{\mathtt{n},\mathtt{t}}(\mathtt{i},\mathtt{j})$ with tiles laid out according to `ord`. Actually, this computation is done by restricting the interleaving of `i` and `j` to their $\mathtt{n}-\mathtt{t}$ high bits corresponding to non-tile dimensions. The `t` low bits of `i` and `j` identify a row- or column-order position within a tile and are, therefore, returned merely concatenated.

As its name may already suggest, function `unzmorton` implements the inverse of `zmorton`. The application of `unzmorton ~tile_n:t ~tile_order:ord n z` computes the two-dimensional coordinates corresponding to a z-address `z`, with the lower $2 \cdot \mathtt{t}$ bits corresponding to a row- or column-order address within a tile. If the respective arguments are omitted, both `zmorton` and `un_zmorton` use a tile size of 0 and a row-major tile order as defaults.

[4] OCaml allows the specification of functions with labeled, optional parameters. In the type signature of these functions, optional parameters are tagged by a question mark and a unique label, e.g., `?label:int`. Actual arguments for these parameters can be omitted in a function call or given by referring to the respective label, (e.g., `~label:42`).

Because the input and output data type common to all examples is an array of float arrays, all recursive matrix multiplications using quad-trees need an implementation of an appropriate conversion. To reduce redundant code, this is done by implementing the methods `pre` and `post` in a virtual subclass of class `abstr_mmult`.

```
class virtual quadtree_mmult
  (tilesize:int) (size:int) (label:string) =
  object
    inherit [float array] Common.abstr_mmult size label

    val width = 1 lsl size
    method pre a b =
      begin
        let qa = Array.create (width*width) 0.0 in
        let qb = Array.create (width*width) 0.0 in
        List.iter begin fun (q,a,o) ->
          for i = 0 to width * width - 1 do
            let (hij,loj) =
              unzmorton ~tile_n:tilesize ~tile_order:o size i
            in
            q.(i) <- a.(hij).(loj)
          done
        end [qa,a,RowMajor;qb,b,ColumnMajor];
        (qa,qb)
      end

    method post rc =
      begin
        let c = Array.make_matrix width width 0.0 in
        for row = 0 to width-1 do
          let c_row = c.(row) in
          for col = 0 to width-1 do
            let j = zmorton ~tile_n:tilesize size (row, col) in
            c_row.(col) <- rc.(j)
          done;
        done;
        c
      end
end
```

Unlike its parent, class `quadtree_mmult` is monomorphic because it is instantiated with the concrete type `float array`. Though, class `quadtree_mmult` provides implementations for the two conversion methods `pre`, which converts the input matrices to a quad-tree, and `post`, which performs the conversion in the opposite direction, method `process` still remains unimplemented, i.e., virtual. Therefore, each matrix multiplication based on a quad-tree data representation can be realized by an implementation of this method in a subclass.

As OCaml provides this alternative to standard arrays, quad-trees can also be implemented by one-dimensional bigarrays. Again, the need for bigarrays arises from the limitations of standard arrays, which have a maximal length restriction and are difficult to be accessed by an external C module. An additional class `quadtree_bigarray_mmult` implements the conversion from and to bigarray quadtrees, making it available to recursive implementations using this data type. Its implementation is not given here as it resembles the implementation of class `quadtree_mmult`.

3.4 Optimizations by Multi-staged Programming

3.4.1 Loop Unrolling

Section 2.4 on page 26 already shows how MetaOCaml can be used to unroll loops partially at run time. This was done by reconstructing the original loop with larger strides and by replicating the code of the loop body a certain number of iterations. So, for an original loop of m iterations, an unrolling of s loop body instances produces a new loop of m div s iterations. Nevertheless, an additional loop for the residual m mod s iterations has to be added after the first loop.

In the case of matrix multiplication, the domain-specific knowledge allows a simpler and more optimized unrolling. Subject of the optimization is the innermost loop of the iterative matrix multiplication, which can be seen as a dot product of a matrix row of A and a matrix column of B. As the body of this loop consists merely of the accumulation of products, we can unroll these summands to a bigger summation expression instead of reproducing the body as a whole. Furthermore, if we restrict the partial unrolling to $s = 2^t$, the reconstruction of the residual loop can be omitted. This is a consequence of the size restriction (3.1), which says that for any matrix M there is an n such that M has width 2^n. Because the number of iterations of the innermost loop is also 2^n, for $t \leq n$, 2^t is a divisor of 2^n and the number of iterations of the residual loop is

$$m \bmod s = 2^n \bmod 2^t = 0.$$

Unlike the loop optimizations applied implicitly by optimizing compilers, like gcc or icc, our explicit generation of loops using MetaOCaml can be controlled by run-time information. So the maximally reasonable stride width s can be set depending on the size of the instruction cache of the machine it is run on. There are several ways to determine the value of s:

- Setting s according to a command line option or configuration file.
- Preceding the actual calculation with minor run-time tests to get a reasonable value for s.
- Calculating s according to a cost model of the machine.

Generic Unrolling Function

For the implementation of this loop unrolling, a function `gensum` generates the partially unrolled inner loop.

```
1 (* val gensum :
     (('a, int) code -> ('b, float) code) ->
     (('b, float) code -> ('a, unit) code) ->
     ('a, int) code -> ('a, unit) code
  *)
6 let gensum exp assign ub =
  .< let ub_k = ((.~ub + 1) / step) - 1 in
     for k = 0 to ub_k do
       let k_of = k * step in
       .~(
11       let build_summand e i = .< .~e +. .~(exp .< k_of + i>. ) >. in
         let base_summand = exp .< k_of >. in
         let sum = iter_up build_summand base_summand 1 (step-1) in
         assign sum
         )
16   done >.
```

Function `gensum` takes three arguments:

- A function `exp` of type `('a, int) code -> ('b, float)` to generate summand code for a given index code. This summand should consist of code to compute the product $A_{i,k} \cdot B_{k,j}$, as given in Definition 3.1 on page 31.
- A function `assign` of type `('b, float) code -> ('a, unit) code` to generate code which adds the result of the partial sum to the accumulator.
- A value `ub` of type `('a,int) code`, the code to be used as upper bound of the innermost loop.

In lines 10–15, the unrolled loop body is constructed. For generating the summation expression, function `iter_up` offers a side-effect-free, tail-recursive abstraction of an iteration. To point out the analogy to a `for` loop, the first argument of `iter_up` can be viewed as description of a state transformer for a given iteration, while the second argument serves as initial state. The remaining arguments specify the lower and upper bounds.

The arguments for the application of `iter_up` in line 13 are the function `build_summand` and variable `base_summand`. The former describes, for an iteration `i`, how to build a new code expression given the code expression `e` of the previous iteration. The second argument `base_summand` carries the code of the initial summand. In the construction of these two parameters, `exp` is used to create the actual summand code. Finally, the application of `assign sum` generates code for the accumulation of the summation result.

Generating the Loop Nest

Since a matrix multiplication using a transposed operand matrix B seems to be most promising with respect to execution time, this is the only iterative implementation in the series of benchmarks to be subject of unrolling. In order to do the loop unrolling only once, the complete loop nest gets generated as object code.

```
  .< fun n a b -> begin
    let width = 1 lsl n in
    let c = Array.make_matrix width width 0.0 in
4   for i = 0 to width - 1 do
      let c_row = c.(i) in
      let a_row = a.(i) in
      for j = 0 to width - 1 do
        let b_col = b.(j) in
9       .~( let assign rhs = .< c_row.(j) <- c_row.(j) +. .~rhs >. in
            let exp k = .< a_row.(.~k) *. b_col.(.~k) >. in
            gensum exp assign .< width - 1 >. )
      done
    done;
14  c
  end >.
```

Note the construction of the inner loop in lines 9–11. It is the application of `gensum` in line 11 which generates the partially unrolled loop, using the previously defined functions `exp` for generating summand code and `assign` for generating assignment code.

To demonstrate the effect of calling `gensum`, the following code is produced for the innermost loop for `step` $= 4$. The code generation inserts this fragment, replacing lines 9–11 of the previous code fragment.

```
  let ub_k = (((width - 1) + 1) / 4) - 1 in
  for k = 0 to ub_k do
  let k_of = k * 4 in
  c_row.(j) <- c_row.(j) +.
5    (a_row.(k_of) *. b_col.(k_of) +.
      a_row.(k_of + 1) *. b_col.(k_of + 1) +.
      a_row.(k_of + 2) *. b_col.(k_of + 2) +.
      a_row.(k_of + 3) *. b_col.(k_of + 3))
  done
```

As OCaml offers two kinds of arrays, it may seem interesting to compare the implementation which uses bigarrays to that using standard arrays. Unfortunately the generated object code using bigarrays turned out to be much slower than the original code without unrolling. Apparently, run-time generated native code currently does not undergo the same optimizations as meta-code: function calls in second-stage code to the bigarray library are not inlined, so each array access is performed by an invocation of the respective external C routine. As this lack of implicit optimization foils the effect of unrolling, an implementation has been considered irrelevant for the benchmark suite.

3.4.2 Offshoring

The environment for offshoring OCaml code to C or Fortran reuses some features of the MetaOCaml implementation. Object code is constructed using the same code brackets (`.< >.`) and the escape operator (`.~`). Like in MetaOCaml, this code generation is performed at run time. Unlike MetaOCaml, the language of the object code is reduced to those OCaml syntax elements which have an equivalent counterpart in C or Fortran. Nevertheless, offshoring code is typed by a static two-stage type system, integrating

the second-stage object code into the first stage meta-code.

To give a simple illustration of this mechanism, consider the following object code, which consists of the loop nest for the iterative matrix multiplication with a transposed operand matrix B.

```
let loop_code = .< fun (a,b,c, width) -> begin
  for i = 0 to width - 1 do
    for j = 0 to width - 1 do
      for k = 0 to width - 1 do
        c.(i).(j) <- c.(i).(j) +. a.(i).(k) *. b.(j).(k)
    done
  done
  done; 4
end >.
```

This implementation uses a dummy return value of 4 because the current implementation of offshoring expects `int` as return type of code objects. Also, note that no compile-time check is performed whether a code conforms to the reduced syntax. Due to the common execution environment which offshoring shares with MetaOCaml, it is not clear at this point whether our object code is going to be offshored or run with MetaOCaml. MetaOCaml does not imply any restriction to the OCaml syntax.

Like MetaOCaml, offshoring provides a special operator `.!{Trx....}` for running second-stage code. The application of this operator initiates the actual offshoring process, which consists of the following tasks.

- Checking the OCaml object code for syntactic compliance and transforming it to C/Fortran code.
- Extending this code with additional marshalling code to perform the translation of OCaml values to C/Fortran and vice versa.
- Compiling the offshored code as separate module by a C or Fortran compiler.
- Dynamically linking the module to the initiating OCaml program.

For our matrix multiplication code

```
let f = .!{Trx.run_gcc} loop_code
```

generates a C module containing the following code.

```
int procedure(double **  a_1, double **  b_2,
              double **  c_3, int width_4 )
{
  int i_5 ;
  for(i_5 = 0; i_5 <= width_4 - 1; i_5++)
  {
    int j_6 ;
    for(j_6 = 0; j_6 <= width_4 - 1; j_6++)
    {
      int k_7 ;
      for(k_7 = 0; k_7 <= width_4 - 1; k_7++)
      {
        (c_3[i_5])[j_6] = (c_3[i_5])[j_6]
                        + (a_1[i_5])[k_7] * (b_2[j_6])[k_7];
```

```
          }
        }
      }
      return 4;
19 }
```

As can be seen in the function head, the OCaml parameters of type `float array array` now appear as C parameters of type `double **`. The structure of this code fragment resembles that of the original OCaml code. The only difference is the enumeration of variables which is a relict of the disambiguation of variables during code generation.

Two implementations using offshoring were added to the benchmark suite. Both are variants of matrix multiplication implementations presented before: one performs an iterative multiplication with a transposed operand matrix B (`IterTrp OAa+O`), the other one is a derivation of the recursive, tiled Strassen algorithm (`RecStr OA+O`). In the latter implementation, the complete iterative tile-level matrix multiplication is extracted as separate code object, which gets offshored in the pre-processing stage.

3.5 Benchmarks

The issues discussed in the previous sections yield a big variety of matrix multiplication implementations, which are summarized in Table 3.1 on page 36 and Table 3.2 on page 52. In order to evaluate the effect of our optimizations, all implementations were rated by an intensive series of benchmarks.

The benchmarks were run on the uniform environment of our hpcLine Linux cluster. Each node of this cluster consist of a 1 GHz Dual-Pentium III with 512 MB of memory. Each processor is equipped with a cache of 256KB. The operating system running on each node is Fedora Core 1 Linux.

Except for the programs using offshoring, the programs were compiled with the native compiler of a developer version based on MetaOCaml 3.07, because the native code compiler of the latest official release, 3.08 alpha 026, contained a deficient array allocation. Only the examples using offshoring were compiled with this release, as the older release did not provide offshoring, yet. For all benchmarks, the timing functions of the MetaOCaml module `Trx` were used. They provide a reliable measuring of execution times by iterating sufficiently often to eliminate short-time effects.

All C code, including the offshored code, was compiled with gcc using the optimization flag `-O3`.

3.5.1 Unsafe Array Access

The negative effect of boundary checks on the overall running time can be seen in Table 3.3. The running times of all implementations using OCaml arrays show a considerable speedup if boundary checks are disabled. In the case of an ordinary iterative matrix-multiplication, arrays of arrays gain most from the use of unsafe array opera-

implementation	time in msec. with bound check	time in msec. without bound check	speedup
`IterOrd OAa`	265.255	98.990	2.68
`IterRow OA`	215.092	194.454	1.11
`IterTrp OAa`	54.217	44.698	1.21
`IterTrp OAa+U`	52.453	37.290	1.41
`RecOrd OA`	47.344	32.172	1.47
`RecStr OA`	25.078	17.690	1.42
`RecStr OA+U`	19.581	13.718	1.43

Table 3.3: Effect of unsafe access on OCaml Arrays for matrix width of 2^{10}

tions (`IterOrd OAa`). this implementation, which was given in Section 3.2.1 on page 37, performs the following array accesses for matrices of width w:

- $2 \cdot w$ accesses for extracting rows of matrix A and C.
- $w \cdot w$ executions of the innermost loop body, each consisting of
  - 2 reading accesses on a row of A and C,
  - 2 reading accesses on B,
  - 1 writing access on C.

This makes a total of $2 \cdot w + 5 \cdot w^2$ individual boundary checks.

All subsequent benchmarks were run with disabled boundary checks. This unsafe use of arrays is usually not recommended, but we can argue with the demand for high performance and the clear computation structure of the matrix multiplication.

3.5.2 Optimal Tile Size

The first series of benchmarks concentrated exclusively on recursive matrix multiplications, as these implementations can be parameterized with a tile width at which the recursion ends and the computation switches to an iteration. The intention of this series was to get an optimal tiling to be used in subsequent benchmarks.

Table 3.6 on the next page displays the execution times of the recursive implementations for tile widths of 2^4 up to 2^8. The two graphs, one for matrices of width 2^9 and one for width 2^{10}, exhibited similarities in the relative execution times: all implementations perform best running with a tile width of 2^7.

3.5.3 Comparison of all Implementations

The main series of benchmarks was run on the complete set of implementations. Since the width of a matrix is always given as the n-th power of 2, values of $n = 7$ up to 11

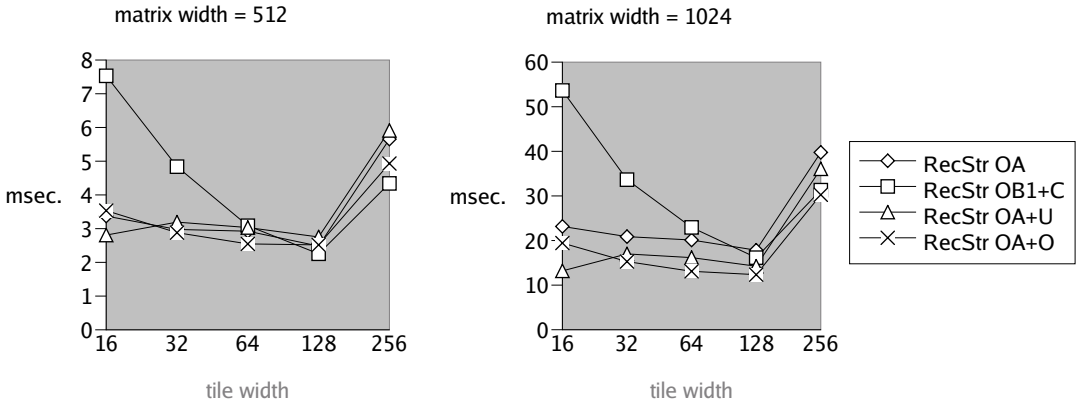


Figure 3.6: Execution times for different tile widths

were used. As a consequence of the size restriction of OCaml arrays, no benchmarks using the type `float array` as data representation could be run for $n = 11$ on the 32 bit i386 architecture.

The tile width was set according to the outcome of the initial benchmarks, i.e., $2^7 = 128$. For the implementations performing unrolling an extent of $2^5 = 32$ replications of the loop body was used. This number is big enough to minimize the overhead of loop bound checks and small enough to make the generated loop body fit into the instruction cache.

The complete benchmark results can be found in Appendix A on page 103. Times are given for the pre- and post-processing as well as for the actual matrix multiplication in the processing phase.

One observation is that the overall execution time of each implementation is dominated by the processing phase. This is within the scope of our expectations because the pre- and post-processing phases have a running time of $O(N^2)$ for matrix width N. This is well below that of the processing phase of $\Theta(N^3)$ or $\Theta(N^{\log_2 7})$. The only exceptions are the implementations using offshoring and unrolling. Here, the generation of code plays a role because it is part of the preprocessing phase. In both the iterative implementations and the recursive implementations this overhead dominates the execution for matrix width of up to 2^8 but it breaks even at about 2^9.

To be able to value the presented optimizations, the charts in Figure 3.7 on page 63 compare the various implementations for matrix width of 2^{10} and 2^{11}. The illustrations collocate the relations between the following three aspects:

- The six algorithmic solutions which make up the benchmark suite (horizontal blocks).
- The six implementations, varying in the use of languages and language features (column colors).
- The execution times (vertical axis).

A missing column indicates either a missing implementation due to a useless configuration, or a missing run due to the range restriction of OCaml arrays.

Results of Iterative Implementations

Comparing both implementations based on arrays of arrays, the C implementation beats the OCaml implementation by a factor of more than 3. The greatest contribution to the good result of the C implementation can be attributed to the cache-aware memory allocation using `posix_memalign()`. Not shown in the diagram here, the exchange of the memory allocation using `malloc` with this function caused a reduction of the execution time by nearly 50 percent (from 29 to 53.5 seconds). Unfortunately, this kind of optimization is not easily adoptable in OCaml unless we would modify the run-time environment used by the compiler.

Surprisingly, all implementations using row-major ordered arrays performed worse than their counterparts using arrays of arrays. For C arrays, the use of cache line aligned allocation hardly had any effect on the overall running time. Both OCaml implementations, one with standard arrays and the other with bigarrays, have a similar execution time. This is due to the fact that the OCaml compiler applies the same optimizations on them, namely the inlining of array accesses and the unboxing of float values. Still, both OCaml implementations are about three times slower than the C implementation.

Two-dimensional bigarrays do not perform any better than one-dimensional ones. Nevertheless, they are a good choice if they are used as interface to an external C program. As the overall execution time of this implementation is not worse than that of a pure C program, the combination of OCaml, bigarrays and external C seems quite attractive.

Changing the implementation to the use of a transposed operand matrix B (which is dual to saying B is column ordered) had a most positive impact on almost all implementations. The only exception is the pure C implementation, which did not gain enough to beat the implementation using arrays of arrays. Nevertheless, in this implementation, unrolling with MetaOCaml as well as offshoring show their potential. In terms of execution time, both are on par with the C implementation.

Results of Recursive Implementations

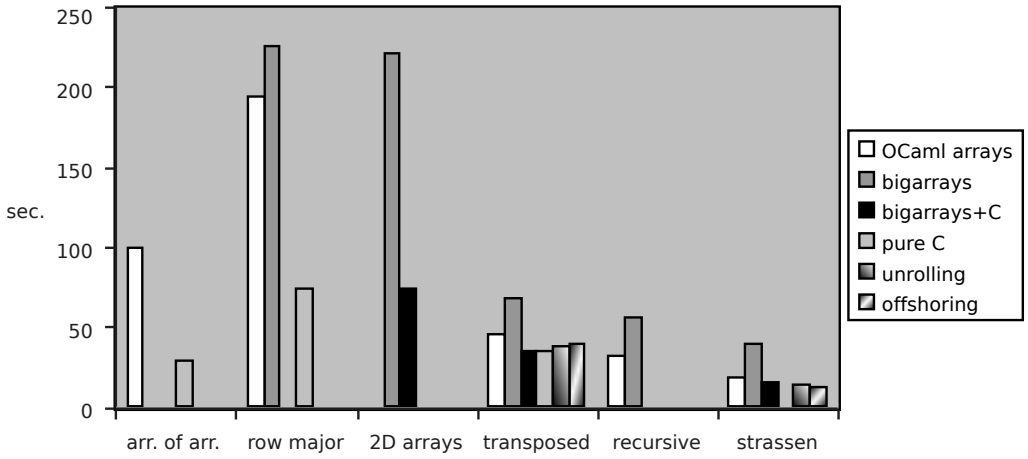
The ordinary recursive matrix multiplication for both OCaml arrays and bigarrays shows little speedup compared to the fastest iterative implementation. Because of the error-prone implementation, which uses complicated index arithmetics to access submatrices, it is questionable whether the efford is justified at all. These arithmetics are needed to avoid both the allocation of memory for intermediate results and the combination of sub-matrices by a copying process.

Strassen's matrix multiplication achieved the best benchmark results. Especially the use of OCaml arrays, with or without unrolling, had a shorter running time than any iterative implementation. Offshoring seems to provide the best from both worlds.

For arrays of width 2^{10}, it beats all other implementations though it contains not a single line of C code. For $n = 11$, where OCaml arrays are no more applicable, the combination of bigarrays and the external tile multiplication in C pays off.

Note that the problem of the length limitation for OCaml arrays is alleviated for 64 bit Intel architectures, where the length of arrays is limited by $2^{54} - 1$. This limit would allow arrays with a size of up to several petabytes.

Matrix width of 2^{10}:



Matrix width of 2^{11}:

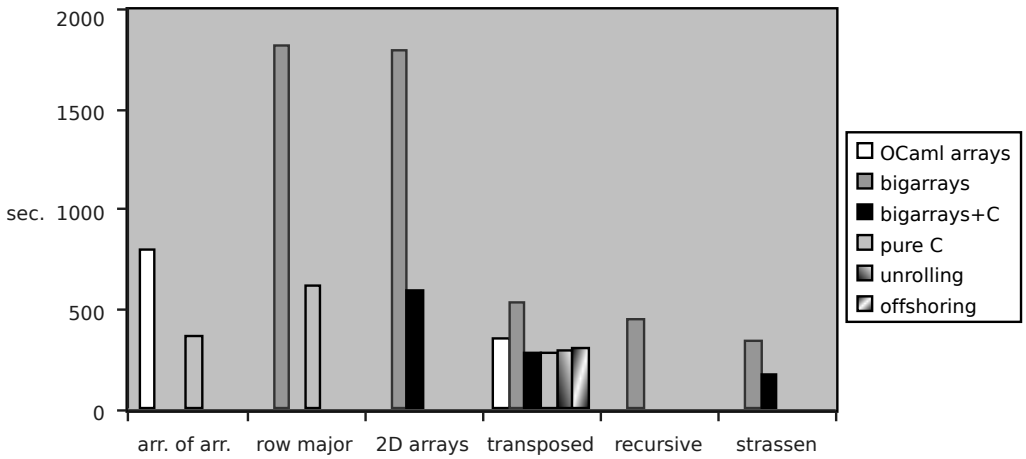


Figure 3.7: Execution times of matrix multiplication.

Chapter 4

Case Study: Image Processing

In contrast to general-purpose languages, like C++ or JAVA, domain-specific languages (DSLs) are tailored to the needs of a special application domain. Well known examples are SQL, the database query language, or XSLT for processing and transforming XML documents. The need for domain-specific languages arises where a general-purpose language fails to meet the demands of the respective domain. For SQL it is the abstraction of the relational database model, for an XSLT program the ability to get parsed by a validating XML-parser. As was the case for SQL standards prior to SQL:1999, a domain-specific language need not necessarily be Turing complete.

For SQL or XSLT, a variety of efficient implementations exist. Other domain-specific languages have only a small community of application programmers which cannot afford the development of highly optimized compilers. Lex and Yacc help to reduce the effort of implementing an efficient compiler front end. For the back end, an interpretation of the abstract syntax tree (AST) often obviates the need for the implementation of a complex code generator. Still, this approach is insufficient for many domains because the interpretation is slower than the direct execution of native code.

Staging an Interpreter Using MetaOCaml

Multi-staged programming is a useful tool for bridging the gap between the ease of implementing an interpreter and the need for compilation. Taha (2004) illustrates this technique with MetaOCaml for a simple example language. Given a program as an AST, the code generator combines recursively OCaml code objects which correspond to nodes in the syntax tree. A special environment is passed to recursive calls of the interpreter in order to map local bindings of variable names to code objects containing the corresponding variables.

The code generation function is also called a *staged interpreter*. This is due to the fact that stripping all staging annotations from this function yields an interpreter for the subject language, which has the same semantics as its staged counterpart. Starting from an interpretation function of a domain-specific language, this implies a natural strategy for writing the code generator by adorning the interpreter with staging annotations.

```
1 let m = [ 0.25 1.0 0.25
          | 1.0  3.0 1.0
          | 0.25 1.0 0.25]
  in
  [ 3 channels:
6 0.125 * sum i from 0 to 2 of
            sum j from 0 to 2 of
              m[i,j] * image(row+i-1, col+j-1, current)
  ]
```

Listing 4.1: Specification of a blur filter.

As a staged interpreter returns an OCaml code object, running this *residual program* can be expected to be considerably faster than the original interpretation. Still, the generation of code and the subsequent compilation with the run operator `.!` require a certain amount of time, too. Only an iterated call of the residual function object can amortize these run-time costs. An indicator for the profitability of staging is the number of iterations at which the staged version breaks even with the iterated interpretation: it should be as small as possible.

4.1 A Domain-Specific Language for Image Processing

Herrmann and Langhammer (2005) present a domain-specific language for the domain of image filtering and show how a staged interpreter, combined with a preceding simplification phase, can speed up the execution of the image processing. This optimization technique will be subject of this case study.

We call the domain-specific specification language to be implemented *subject language*. MetaOCaml, the language chosen for implementation, is called *base language*. Accordingly, the implementation of the subject language is called *base program*.

The subject language presented is not Turing complete as it lacks the concept of general recursion. Nevertheless, it provides constructs for summation, local and non-local pixel access and matrices of float constants.

An expression in the subject language defines each output pixel by referring to a number of pixels of the input image. As main target application, this language provides also a convenient way of specifying image filtering tasks based on convolution. To give an impression, consider the example specification of a blurring filter in Listing 4.1 and the effect on input images shown in Figure 4.1 on the next page.

Like all filter specifications, this blurring filter describes how to compute the output pixel at coordinate `(row,col)` by an expression which refers to a number of pixels in the input image. The filtering process iterates the application of this filter for each coordinate of the output image.

In this example, the `let` binding defines variable `m` to be a convolution matrix of float values. The summations in lines 6–8 specify the intensity of the output pixel by referring to pixels in the neighborhood of the respective coordinate in the input image. The head of the block `[ 3 channels:` *exp* `]` indicates that the same expression is

Figure 4.1: Sample image before and after applying blur filter

used for each of the three color channels `red` $= 0$, `green` $= 1$ and `blue` $= 2$ of the output image. Variable `current` gets bound to the number of the respective output channel in order to provide a comfortable way to refer to the corresponding channel of the input image. If the filter specifications for the channels differ substantially, the alternative syntax of a semicolon-separated list `[` exp`;` exp`;` exp `]` can be used instead.

4.2 Optimization Using Staged Programming

The potential for optimization is evident if we consider the filter given above to be executed by an interpreter. As the filter specification refers to a single pixel, for an input image of size 1024×768 it would have to be interpreted 786432 times. For each pixel, the interpreter would create a convolution matrix and reproduce the summations by 9 iterations accumulating the result.

The following two conceptually independent optimization techniques are introduced in order to reduce this effort.

1. By adding staging annotations to the interpreter we can build a simple compiler. Figure 4.2 on the following page without the shaded elements shows the original interpretation, which yields a residual filter function by the residual function generator. No simplification is done at this point, as this corresponds to a simple partial application. Adding staging annotations makes the residual function generator produce object code. Now, the AST is analyzed to generate object code, postponing the actual execution. This code generation yields residual object code, which is transformed into a OCaml function by calling the run operator.

2. Certain expressions in the subject program depend exclusively on information known before the actual filter execution, like the number of summands in List-

ing 4.1. These expressions can be reduced in advance, reducing their repeated evaluation during the filter application later on. This optimization technique incorporates a binding time analysis in order to determine which code fragments depend entirely on *static* input, i.e., the part of the input known at the time of the program specialization. All other input, which is unknown until run time, and all expressions depending thereon are called *dynamic*. The binding time analysis is combined with an immediate simplification of static expression, a technique for the automated specialization of programs which is known as *online partial evaluation* (Jones et al., 1993, Chapter 7).

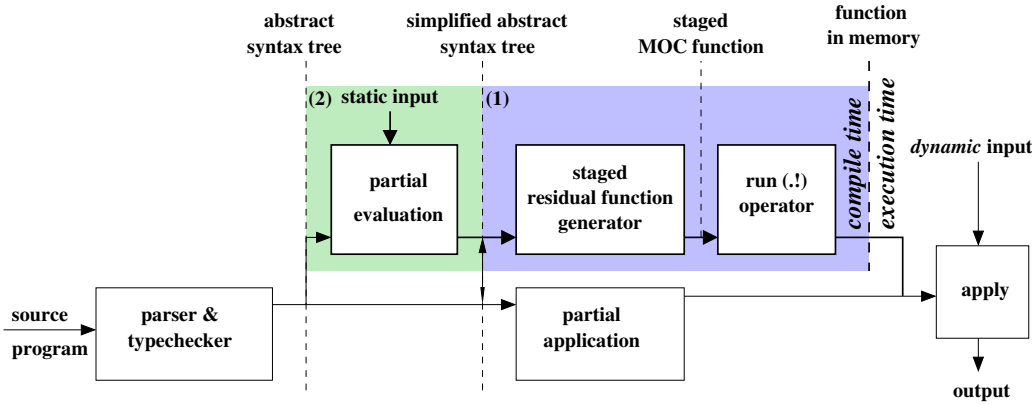


Figure 4.2: Design of the DSL implementation. The shaded components substitute a pure partial application with the optimization by a partial evaluation and the staging of the interpreter

4.3 Language Design

Because we describe the design of the image filtering language in detail elsewhere (Herrmann and Langhammer, 2005), only a brief overview is given here.

4.3.1 Syntax

A filter specification consists of a top-level expression (*toplevelExpr*), which is composed of a series of local bindings common for all color channels and a list of sub-expressions (*channelSpecs*) for specifying the intensity for the each color channel.

$$\textit{toplevelExpr} \leftarrow (\texttt{let}\ \textit{var}\ \texttt{=}\ \textit{expr}\ \texttt{in}\ \ \textit{toplevelExpr})\ \textit{channelSpecs} \tag{4.1}$$

$$\textit{channelSpecs} \leftarrow \texttt{[}\ \textit{expr}\ (\texttt{;}\ \textit{expr})^{*}\ \texttt{]} \tag{4.2}$$

$$\mid \texttt{[}\ \textit{constInt}\ \texttt{channels}\ \texttt{:}\ \textit{expr}\ \texttt{]} \tag{4.3}$$

The standard form (4.2) of the channel specification is enclosed in brackets and adjacent entities are separated with a semicolon. As stated before, using the alternative (4.3) with a single expression *expr* is a convenient abbreviation. It is expanded during an initial desugaring process: *constInt* copies of *expr* are enclosed in a `let` expression binding variable `current` to the respective channel number and sequenced in a semicolon-separated channel list.

The right hand side of the common `let` bindings and the elements of the channel specification list are expressions (*expr*) of the same kind.

$$
\begin{aligned}
\mathit{expr} \leftarrow\ & \mathit{const} && (4.4)\\
|\ & \mathit{var} && (4.5)\\
|\ & \texttt{image (} \mathit{expr} \texttt{ , } \mathit{expr} \texttt{ , } \mathit{expr} \texttt{)} && (4.6)\\
|\ & \texttt{(} \mathit{expr} \texttt{)} && (4.7)\\
|\ & \mathit{prefix1}\ \mathit{expr} && (4.8)\\
|\ & \mathit{prefix2} \texttt{ (} \mathit{expr} \texttt{ , } \mathit{expr} \texttt{)} && (4.9)\\
|\ & \mathit{expr}\ \mathit{infix}\ \mathit{expr} && (4.10)\\
|\ & \texttt{if } \mathit{expr} \texttt{ then } \mathit{expr} \texttt{ else } \mathit{expr} && (4.11)\\
|\ & \texttt{let } \mathit{var} \texttt{ = } \mathit{expr} \texttt{ in } \mathit{expr} && (4.12)\\
|\ & \texttt{sum } \mathit{var} \texttt{ from } \mathit{expr} \texttt{ to } \mathit{expr} \texttt{ of } \mathit{expr} && (4.13)\\
|\ & \mathit{matrix} && (4.14)\\
|\ & \mathit{index} && (4.15)
\end{aligned}
$$

Some of these production rules are similar to the grammars of other languages, like constants, variables, infix and prefix operators.

$$
\begin{aligned}
\mathit{const} &\leftarrow \mathit{constBool} \mid \ldots && (4.16)\\
\mathit{var} &\leftarrow \mathit{letter}(\mathit{letter} \mid \mathit{digit})^* && (4.17)\\
\mathit{prefix1} &\leftarrow \texttt{-}_1 \mid \texttt{not} \mid \texttt{floor} \mid \texttt{sin} \mid \ldots && (4.18)\\
\mathit{prefix2} &\leftarrow \texttt{min} \mid \texttt{max} && (4.19)\\
\mathit{infix} &\leftarrow \texttt{+} \mid \texttt{-}_2 \mid \texttt{*} \mid \texttt{<} \mid \texttt{\&\&} \mid \ldots && (4.20)\\
\mathit{constBool} &\leftarrow \texttt{false} \mid \texttt{true} && (4.21)
\end{aligned}
$$

Also, prefix operators have a higher precedence than infix operators and the precedence among infix operators is defined according to the common convention in mathematics. Note that there are two minus operators: a unary prefix $\texttt{-}_1$ and a binary infix $\texttt{-}_2$.

The only non-standard expressions are summation (4.13) and matrix expressions, which are restricted to two dimensions and float constants as matrix elements.

$$
\begin{aligned}
\mathit{matrix} &\leftarrow \texttt{[} \mathit{constFloat}^+ \ (\texttt{|}\ \mathit{constFloat}^+)^* \texttt{]} && (4.22)\\
\mathit{index} &\leftarrow \mathit{var} \texttt{ [} \mathit{expr} \texttt{ , } \mathit{expr} \texttt{]} && (4.23)
\end{aligned}
$$

4.3.2 Typing

The specification language provides the types int, float and bool, which are inferred by semantic actions of the parser, according to the typing rules given by Herrmann and Langhammer (2005).

4.3.3 Semantics

Only a few characteristics of the denotational semantics are given here. An environment ε is used to manage the binding of variables to values, where $\varepsilon(v)$ denotes the value assigned to variable v.

No Turing completeness. In favor of a higher degree of optimization, the language does not feature general recursion. Therefore, fixed-point semantics is not required.

Operator overloading. The same arithmetic operators are used for float and int values. If for a binary operator both operands are of the same type, the respective operation for that type will be used. If both int and float appear in the same operation, the int value will be automatically coerced to the respective float value. The coercion of int values is also done for operations on float values. For conversions in the opposite direction, operator `floor` must be used to make the programmer aware of the loss of precision.

Image indexing. For a pair of coordinates *row* and *col* and a color channel *ch*, given as integer expressions, the expression `image(`*row*`,`*col*`,`*ch*`)` returns the color intensity of the respective pixel as a float value in the range of $[0.0, 1.0[$. For the convenience of the programmer, the channel numbers are named by meaningful, predefined variables ($\varepsilon(\texttt{red}) = 0$, $\varepsilon(\texttt{green}) = 1$, $\varepsilon(\texttt{blue}) = 2$ and $\varepsilon(\texttt{gray}) = 0$). Of course, the coordinates must be within the dimensions of the input image, i.e.,

$$(row, col) \in \overline{\varepsilon(\texttt{height})} \times \overline{\varepsilon(\texttt{width})}$$

Summations. The summation expression (`sum k =` *lo* `to` *hi* `of` *body*) has the semantics of the summation symbol in mathematics. For each integral point *i* in the range of $\{lo, \ldots, hi\}$, the environment is extended with a binding $\varepsilon(\texttt{k}) := i$ and used to evaluate *body*. The results are accumulated and returned as result of the complete summation.

4.4 Language Implementation

4.4.1 Data types

For the front end of the compiler/interpreter, the tools ocamllex and ocamlyacc were used to scan and parse an image processing specification. As result and for the internal representation of the subject program, record data types are used to represent the abstract syntax.

```
type exp     = { dtype:dtype; op:op; args:exp list; expand:int }
type top_exp = { common_bind: (string * exp) list; channels: exp list }
```

The root node of an AST is of type `top_exp`, which is a record containing a list of common bindings and a list of specifications for each color channel. This record resembles the common `let` bindings of the production rule for *toplevelExpr* (4.1) and the channel list of the first production rule of *channelSpecs* (4.2). As stated before, the alternative, abbreviated form (4.3) gets expanded to a channel list by the desugarer.

A base expression `exp` consists of a record with the following fields:

- a data type `dtype`, which is set by the type inference system implemented in the semantic actions of the parser,
- an operation `op`, which corresponds to one of the production alternatives, (4.4) to (4.15) of *expr*,

```
type op =
    C of value  | V of string | Image  | Int2Float | Floor
  | UnOp  of unOp | BinOp of binOp | If | Let of string
  | IndexMatrix of string | Sum of string
```

- a list `args` of sub-expressions and
- an expand value `expand`, which is used to guide the loop unrolling optimization.

OCaml sum types are used for the representation of both data types and values of these data types.

```
type dtype = Bool | Int | Float | Matrix
type value = VInt of int | VFloat of float | VBool of bool
           | VMatrix of float array array
```

In order to reduce the need for additional pattern matching, the data types for unary and binary operators use alternatives for each type of the subject language.

```
type unOp  = NegI | NegF | Not | AbsI | AbsF ...
type binOp = AddI | SubI | MulI | DivI | PowI | MinI | MaxI | ModI ...
```

E.g., `AddI` is used for an int summation, `AddF` a float summation.

4.4.2 Expression Simplification

For simplicity, the program specialization performed on image filter expressions is an *online partial evaluation*, which indicates that binding time analysis (BTA) and simplification are not separate phases but are intertwined to a single simplifying function. The advantage over a separate BTA and simplification phases is a much simpler specialization, as static expressions get instantly simplified to a single value and, thus, can be used to guide the subsequent analysis. The typical problem of non-termination is not an issue for this online partial evaluation because the subject language has no general

concept of recursion. Also the code explosion by "over-specializing" a `sum` expression is prevented by a partial loop unrolling.

Two functions form the main entry points of the simplification process.

```
val simplify_top_exp : top_exp -> value environment -> top_exp
val simplify_exp : exp -> value environment -> exp
```

Function `simplify_top_exp` takes the root of an AST and an environment of static values. The initial environment is preset with bindings for all static parameters, like `width`, `height`, `red`, `green`, ... For sub-expressions, the simplification is delegated to `simplify_exp`. The general principle of simplifying an expression can be summarized as follows:

- Simplify sub-expressions recursively.
- Depending on which sub-expressions could be reduced, i.e., which of them are merely constant values, a simplification step is performed for the current node. In the best case the respective operation of the current AST node is instantly performed, in the worst the AST node has to be reconstructed.

Function `unC`, which is of type `exp -> value option`, provides a convenient checking mechanism: if the AST node `e` contains only a constant value `c`, it returns `Some c`. Otherwise, `None` is returned, which indicates that `e` is a non-constant expression. So, if `e` is a sub-expression which is a result of a recursive simplification call, a pattern match on `unC e` tests whether e is completely reduced or an expression which contains irreducible expressions. In terms of binding time, e is static in the former case and dynamic in the latter.

To give an example, in the case of a multiplication operator the following code fragment performs the simplification.

```
  | MulI | MulF ->
      begin match unC a, unC b with
3     | Some av, Some bv -> exp_of_value (binopfun op av bv)
      | Some av, _ when is_zero av -> zero_of dtype
      | _, Some bv when is_zero bv -> zero_of dtype
      | Some av, _ when is_one av -> b
      | _, Some bv when is_one bv -> a
8     | _ -> combine a b
      end
```

Here, the matching is done on the expressions of the two simplified operands. The most successful case is in line 3, as a new constant value expression can be generated by multiplying both constant operands. The cases in lines 4–7 perform minor optimizations, namely $0 \cdot a \rightarrow 0$ and $1 \cdot a \rightarrow a$. If only the last case matches, the current node is reconstructed by calling `combine` because this operation cannot be simplified any further.

Simplifying `let` expressions is a slightly more tricky issue, but follows the same principle. First, the right hand side expression is simplified recursively. In the case of a complete reduction, the constant returned is added to the static environment and used to simplify the body. If the right hand side is dynamic, the body is simplified with the

old environment. If the simplified body is a constant, it is returned, otherwise the `let` expression is reconstructed.

The simplification of a variable amounts to a lookup in the environment. If the lookup is successful, the constant value is returned. Otherwise, the variable is returned unchanged.

Partial loop unrolling is done for summation expressions if the size of the resulting code does not exceed a certain limit M, given as command line argument. The size of an expression is given by the `extent` value in its AST node, which, in fact, indicates the number of leaves in the respective subtree. For a partial unrolling of a sum which has a body with an extent of e, the number of replications N of the body is limited by

$$N \cdot e < M.$$

As a consequence, N is set to $\lceil M/e \rceil - 1$ and loop unrolling is performed only in the case that $N > 1$.

The degree of simplification performed on a sum also depends on the binding times of its sub-expressions. E.g., in the case of static bounds and a static body, the whole `sum` expression is completely reduced. For static bounds with less then N iterations and a dynamic body the loop unrolling is complete rather than partial. This situation is met for the nested summation of the blur filter in Listing 4.1 on page 66.

4.4.3 Code Generation

As mentioned before, the generation of code is done by a staged interpreter. All respective operations are protected against evaluation by code brackets and, as results of recursive interpretation are now returned as code objects, the escape operator is used to combine sub-expressions to a new code fragment. As the type of the returned code depends on the type of the respective AST node, a sum type is introduced.

```
type 'a codevalue = CInt of ('a,int) code
                  | CFloat of ('a,float) code
                  | CBool of ('a,bool) code
                  | CMatrix of ('a,float array array) code
```

Function `lift`, which is of type `value -> 'a codevalue`, transforms a value into the respective codevalue.

Entry point for the code generation is function `codegen_top_exp` which takes a top-level expression and an initial environment of static variables, as well as the input image.

```
val codegen_top_exp :
  top_exp -> 'a codevalue environment ->
  image -> ('a, int list) code
```

Note that no simplification or evaluation of expressions is done during code generation. All optimizations have already been performed in the partial evaluation phase before. Consequently, the values managed in the environment consist exclusively of code containing a variable. This is due to the fact that `let` nodes, like

`let i = ... in` e, are transformed directly into the corresponding OCaml `let` expression `.< let k = ... in ... >.` by adding the binding `"i"` $\rightarrow$ `.< k >.` to the environment before generating the body code.

To demonstrate the effect of staging annotations in the interpreter, consider the generation of loop code for a summation expression.

```
  | Sum s ->
2   let (lb,ub,body) = head3 args in
    let lb' = unCInt (codegen_exp lb env source)
    and ub' = unCInt (codegen_exp ub env source) in
    let bodycode i =
      let env' = extEnv (s,CInt i) env in
7     codegen_exp body env' source
    in
    let dynFor low high init step =
          .< let s = ref .~init in
             for i = .~low to .~high do
12              s := .~(step .<i>. .<!s>.)
             done;
             !s
          >.
    in
17  begin match dtype with
    | Int   ->
        let step i s = .< .~s +  .~(unCInt (bodycode i)) >. in
        CInt (dynFor lb' ub' .<0>. step)
    | Float -> ...
```

In lines 2–4, the sub-expressions are extracted from the body list and code for both summation bounds is generated recursively. The definition of a local function `bodycode` follows in lines 5–8, which generates code for the body using the iteration variable set in parameter `i`. Function `dynFor` in lines 9–16 is a polymorphic function used to construct the loop nest. The actual code generation is performed depending on a pattern match in line 17 on the data type of the node. Function `step` defined in line 19 generates code for a single addition and is used as parameter to create the body of the loop.

To motivate the notion of a staged interpretation, if we stripped all MetaOCaml syntax from the code above, i.e., if we substituted `.~` with the empty string and `.< ... >.` with `( ... )`, the remaining code would form an interpretation of the summation expression. All code that was staged before would now be executed immediately, including the iteration to evaluate and accumulate the summands.

4.5 Optimization by Example

After scanning and parsing of the expression in Listing 4.1 on page 66, a type-annotated AST is the subject of all subsequent optimizations. To give an impression, consider the following fragments of this data structure.

```
  Defs.top_exp =
  {common_bind =
    [("m", {dtype = Matrix;
           op = C (VMatrix [|[|0.25; 1.; 0.25|];
                            [|1.; 3.; 1.|];
                            [|0.25; 1.; 0.25|]|]);
           args = []; expand = 1})];
   channels =
    [{dtype = Float; op = Let "current";
      args =
       [{dtype = Int; op = C (VInt 0); args = []; expand = 1};
        {dtype = Float; op = BinOp MulF;
         args =
          [{dtype = Float; op = C (VFloat 0.125); args = []; expand = 1};
           {dtype = Float; op = Sum "i";
            args =
  ...
               {dtype = Float; op = IndexMatrix "m";
                    args =
                     [{dtype = Int; op = V "i"; args = []; expand = 1};
                      {dtype = Int; op = V "j"; args = []; expand = 1}];
                    expand = 2}
  ...
```

The AST is a direct equivalent of the input program, as each node has a corresponding syntactic element. For example, the static matrix entries have not been inlined, yet. The matrix is bound to `m` in the section of common bindings and used later by indexing this variable.

The application of the simplification changes this expression dramatically.

```
  Defs.top_exp =
{common_bind = [];
   channels =
    [{dtype = Float; op = BinOp MulF;
      args =
       [{dtype = Float; op = C (VFloat 0.125); args = []; expand = 1};
        {dtype = Float; op = BinOp AddF;
         args =
          [{dtype = Float; op = BinOp AddF;
            args =
             [{dtype = Float; op = BinOp AddF;
               args =
                [{dtype = Float; op = BinOp AddF;
                ...
```

Now, the two summations have been unrolled completely by the replication of the body, substituting both index variables `i,j` with the respective iteration and concatenating the resulting expressions with additions. Also each matrix indexing operation has been replaced by the corresponding matrix entry. The construction of the matrix in the head is omitted.

To get a better impression of the optimizations performed, consider the result of the code generation for the simplified AST.

```
.<fun (row, col) -> Array.of_list [
  let c =
    int_of_float (0.125 *.
      (0.25 *. (float_of_int rast.(row-1).(col-1).(0) /. 255.) +.
                float_of_int rast.(row-1).(col+1-1).(0) /. 255. +.
       0.25 *. (float_of_int rast.(row-1).(col+2-1).(0) /. 255.) +.
                float_of_int rast.(row+1-1).(col-1).(0) /. 255. +.
         3. *. (float_of_int rast.(row+1-1).(col+1-1).(0) /. 255.) +.
                float_of_int rast.(row+1-1).(col+2-1).(0) /. 255. +.
       0.25 *. (float_of_int rast.(row+2-1).(col-1).(0) /. 255.) +.
                float_of_int rast.(row+2-1).(col+1-1).(0) /. 255. +.
       0.25 *. (float_of_int rast.(row+2-1).(col+2-1).(0) /. 255.)) *.
    255.)
  in
  if (c < 0) then 0 else if (c > 255) then 255 else c;
...;...] >.
```

Only the expression for the red channel is shown here; the expressions for the other channels are similar. Note that, aside form unrolling and inlining, further minor optimizations have been performed, like $1 \cdot a \longrightarrow a$. Because the subject language works with floating-point color intensities and the image format uses integer intensities in the range of 0 to 255, both appropriate conversions and the handling of range violations have been added to the code.

4.6 Benchmarks

Herrmann and Langhammer (2005) give more examples as well as benchmarks comparing filter execution done by pure interpretation with those using simplification and staged interpretation.

For the blur filter, a number of additional benchmarks determine both the overhead of the run-time optimizations and the break-even point, i.e., the number of iterations needed to outperform the unstaged interpretation. The benchmark platform was the same as that of the previous chapters, i.e., a 1 GHz Dual Pentium III with 512 MB of memory running Fedora Core 1 Linux. The implementations were compiled with `metaocamlopt` of our developer version based on OCaml 3.07. Unsafe array access was activated. For all runs, the timing functions from the `Trx` module of MetaOCaml were used. By iterating the probe sufficiently often, these timing functions enable the system to give a reliable timing.

The first series of benchmarks investigates how much time is needed to perform the optimizations introduced to the base program. Simplification on the one hand and staged interpretation (i.e., code generation and compilation by `.!`) on the other are independent optimizations. Therefore, they can be activated or omitted by a command line flag when running the image filtering tool. Nevertheless, the running time of the code generation is largely dependent on the outcome of the simplification. Especially the loop unrolling of the blur filter expression inflates the size of the intermediate AST distinctly. For each color channel the loop body is replicated 9 times. This increases the work for generating a code object as well as for compiling it with the run operator.

The execution times given in Table 4.1 demonstrate this effect. Deactivating the simplification reduces the time for the remaining optimizations to about 60 % for the native program and to about 44 % for the bytecode program.

A conspicuous effect can be noted, when comparing the native code compilation of the base program with the corresponding bytecode version. The time for executing the simplification and the code generation in native code is faster than the time for these optimizations in bytecode. As bytecode is interpreted by a virtual machine, this result is what can be expected. The surprise is that the application of the `.!` operator takes noticeably more time if performed for native code. Code compilation and dynamic binding is more complex for native code as it is for bytecode.

base program	native		bytecode	
	simplification		simplification	
optimization	active	inactive	active	inactive
simplify	1.110	–	4.030	–
code gen.	2.282	0.468	5.747	1.417
run (.!)	80.303	49.843	33.637	15.821

Table 4.1: Execution time of optimization in msec.

Table 4.2 shows the execution times of the blur filter for a square input image of width 1000. Results are given for both the native and the bytecode compilation of the base program. The base program was run for all four configurations which are possible by activating or deactivating the two optimizations. As result of the $1000 \times 1000 = 10^6$ applications of the filter expression, the overhead spent for the optimizations turned out to be marginal compared to the overall execution time, and the speedups compared to the respective unoptimized run are enormous. The activation of both optimizations yielded the best speedups.

The good speedups for input images of size 1000×1000 imply that the image dimensions, where the constant overhead equals the gains of the optimized executions must be much smaller than 1000. This *break-even point* is important in order to value the applicability of the optimizations performed.

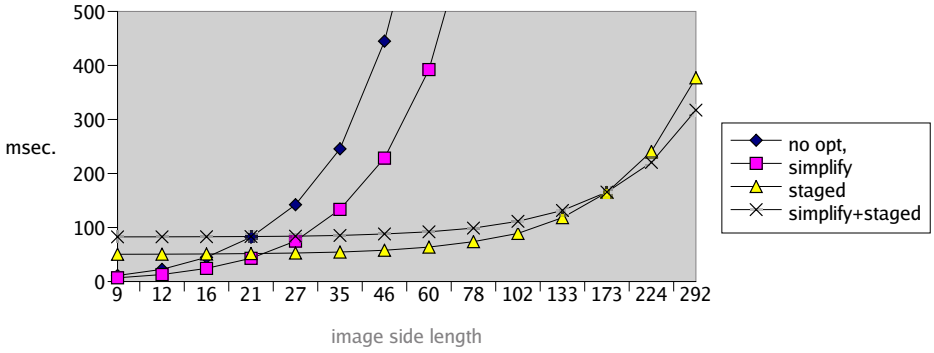
The result of a series of test runs for various image sizes is shown in Figure 4.3 on the next page. Again, all four configurations were benchmarked and the resulting

	native base program		bytecode base program	
configuration	t in sec.	speedup	t in sec.	speedup
none	259.96	1.00×	1 032.33	1.00×
simplify	135.32	1.92×	640.04	1.61×
staged*	3.98	65.24×	29.90	34.52×
simplify + staged*	2.75	94.68×	15.83	65.21×

* Staged run comprises code generation and `.!` application.

Table 4.2: Overall execution times for input image of size 1000×1000.

Native base program:



Bytecode base program:

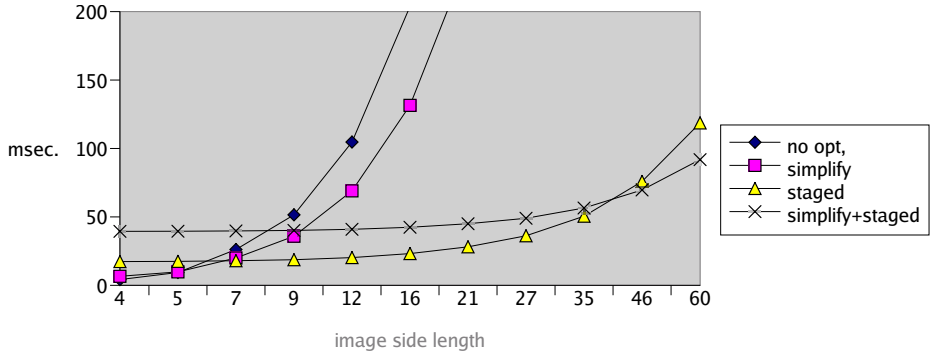


Figure 4.3: Overall execution times for square images of different sizes.

execution times were recorded in the diagram.

Due to the more expensive application of the `.!` operator, the native code base program has to perform more iterations than its bytecode counterpart in order to outperform the image filtering without any optimization. Not until the image width exceeds 21 pixels, all implementations using optimizations are faster than the unoptimized one. Examining the effect of simplification, the break-even point for an unstaged run is below 9×9 pixels. For a staged execution, no positive effect appears until the image size exceeds about 173×173 pixels. This gap is caused mainly by the additional computations the run operator has to perform for the larger, simplified code object.

For the bytecode compilation, the overall break-even point is much smaller due to a reduced optimization overhead. For any combination of optimizations the overhead is amortized for an image width of about 9. Again, the simplification amortizes for much bigger values if it was combined with staging. Though, this break-even point of about 40×40 pixels is smaller than that of the native code compilation, caused by the less expensive run operation for bytecode.

4.7 Conclusions

MetaOCaml provides a high-level tool for optimizing the interpretation of a domain-specific language implementation. For our language, which does not have a concept for general recursion or iteration, an online partial evaluation was quite easy to implement. It is combined with a preceding simplification of the AST, generating additional speedup. As the benchmarks revealed, these optimizations entail a notable overhead of computations, which grows in the size of the AST. In the native compilation, which is the only relevant in terms of high-performance computing, this overhead is bigger compared to the respective bytecode. Fortunately, a repeated execution amortizes the overhead, compared to an unstaged run without code simplification. Because, for our implementation, the number of interpretations grows fast enough with the problem size, the optimizations pay off already for images of GUI icon size.

Chapter 5

Case Study: Parallel Karatsuba Polynomial Product

5.1 Introduction

Parallel computing forms a self-contained special-purpose domain within the field of high-performance computing. The main idea of parallelization is to reduce the overall computation time by separating the number of computation into independent parts, which can be distributing on a sufficiently large number of processors. Lengauer (2004) gives a concise overview of this domain.

For a parallel environment with distributed memory, communications are needed to pass messages between cooperating processes. The need for communication arises from data dependences between computations of different processes. In the development of parallel programs, currently, these communications are often described by point-to-point communications. These communications allow a fine-grained specification of the exchange of data between a pair of processes; one acting as sender, the other as receiver.

As complex communication structures often derive from complex dependencies between computations, the tuning of send-receive communications by hand tends to become unmanageable and error-prone. There are several approaches to overcome this problem. Gorlatch (2004) compares state-of-the-art parallel programming with the unstructured imperative programming style of 25 years ago. Just like unstructured jumps are frowned upon by software engineers because they compromise any high level abstraction of the program flow, send-receive operations should be banned in favor of high-level collective operations. These operations provide a description of the underlying communication pattern, comprehensible for both the implementer and an optimizing compiler. Gorlatch argues that the gain in predictability is at the expense of neither expressiveness nor performance.

This chapter discusses how MetaOCaml can be used for parallel computing. OCaml bindings to the C interface specification of the MPI standard (MPI Forum, 1997) provide machine-independent point-to-point and collective communications. A further

discussion presents another approach by Herrmann (2005), which reduces the dangers of hand-written send-receive communications by introducing layers of abstraction to separate domain concerns. Finally, the example implementation of the polynomial product given by Karatsuba and Ofman (1962) is inspected for further optimization potential.

5.2 Parallel Computing for High-Performance

Amdahl (1967) pointed out, that the degree of parallelism is limited by inherently sequential fragments of the algorithm, which arise from dependencies between computations determining an order of execution in time. Fortunately, there are many applications for which Amdahl's law does not apply. Gustafson (1988) argues, that in many real-life applications, the number of parallel computations is not constant but grows with the resources available. Thus, the amount of time spent on the sequential part of a program becomes less important when the number of processors grows. Also, it is reasonable to consider the execution time as constant and value how the number of parallel computations grows with the number of processes.

In real world implementations the parallel execution is also diminished by factors emerging from the underlying architecture. On machines providing *shared memory parallelism* (SMP) several processes use a common region of memory. In order to prevent update anomalies, exclusive memory access has to be granted for conflicting operations of several processes on the same memory location. Each process wishing to access a datum shared by many is granted exclusive access. In fact, accesses of several processes on the same exclusive memory location result in a sequential execution, whose order is undetermined.

On *distributed memory machines*, like Linux clusters, a parallel program consists of processes, each of which holds its own memory. For the exchange of information, these processes have to communicate with one or a number of other processes. Depending on the architecture, communication is performed via a bus or an interconnecting network. Whatever hardware is used, communications are accountable for a certain overhead in the overall execution time of parallel programs.

In order to create an abstraction from low-level communication protocols, a number of paradigms and frameworks have been developed. The intention of these abstractions is to make programming safer and more predictable as well as to increase the portability of programs. In terms of the application domain, they aim to provide tools to enable the developer to concentrate on solving the problem rather than on taming the machine.

5.3 Message Passing Interface (MPI)

The MPI standard defines an API of communication primitives for implementing message passing parallel programs. The intended programming style, which is commonly used on parallel machines with distributed memory, is also called SPMD, for "single

program, multiple data" (Foster, 1995). Each process in a parallel environment runs the same program but encapsulates its own data. By referring to the identifier of another process (called *rank* in MPI), messages can be sent or received.

The popularity of MPI in the domain of parallel computing is mainly based on the abstraction from underlying, hardware dependent communication protocols (TCP/IP, Myrinet, ...), on the one hand, and on the fine-grained tuning options by single point-to-point communications, on the other.

For exchanging data between processes, MPI provides two kinds of communication abstractions

Point-to-point communication by a set of mid-level functions. These primitives are independent from underlying, hardware-specific communication protocols, like TCP/IP or even shared memory. Which role a process takes in a communication is defined by the application programmer by a case distinction on the process rank.

Collective communications by a set of high-level functions. Each of them provides a certain communication structure, e.g., distributing data from one process to all others or combining distributed data with a given operation in a parallel fashion and providing the result to a single process or all processes.

The advantage of point-to-point communications is the chance to perform a fine-grained tuning of the parallel application. One of the two processes performing a communication acts as sender and the other acts as receiver. Such a send-receive operation can be either blocking or non-blocking. In blocking mode, the sending process does not return until the send buffer has been stored away safely and the receiver waits until the receive buffer has been filled successfully. In non-blocking mode, sender and receiver both return immediately to resume their computation. For further adjustments, MPI provides special communication modes, like *buffered*, *synchronous* and *ready* point-to-point communication (see MPI Forum (1997) for details).

By using collective operations, the application programmer relies on the communication patterns implemented in the MPI library he is using. These libraries often come with efficient implementations which are tailored to the architecture of the respective target machine.

Further features of the MPI 1.1 standard are the definition of communication contexts, process groups and topologies and bindings for both C and Fortran77. MPI 2.0 extends the standard by features like one-sided communications, remote memory access, process management, parallel I/O and new language bindings.

Currently there are a number of implementations, like LAM (Burns et al., 1994) or MPICH (Gropp et al., 1996), which covers the MPI 1.1 API and big parts of MPI 2.0. Custom-built implementations for special high-performance hardware are available, like MP-MPICH (Pöppe et al., 2005), which is used on our SCI interconnected hpcLine cluster.

5.4 OCaml and MPI

5.4.1 OCamlMPI – a Binding to the C Interface.

As most frequently used languages in the domain of parallel computing, only the MPI bindings to C/C++ and Fortran are defined in the standard. Therefore, the OCaml binding OCamlMPI (Leroy, 2001) is closely geared to and uses that of C.

OCamlMPI covers a large subset of MPI, including point-to-point and collective communications. It uses the interfacing mechanism of OCaml to C, which is further discussed in Section 6.2 on page 95. According to this, the organization of the distribution is divided into an interface given in `mpi.mli`, a shallow implementation `mpi.ml` consisting mainly of external declarations, and C stub functions calling the respective primitives of the MPI interface. The comments in the interface file give a brief description of the semantics for each of the functions provided.

In order to minimize the loss of performance, most communication functions come in five flavors.

- a polymorphic function taking any data type (like `MPI.send`)
- four monomorphic functions, specialized for the following types
  - int (`MPI.send_int`)
  - float (`MPI.send_float`)
  - int array (`MPI.send_int_array`)
  - float array (`MPI.send_float_array`)

The polymorphic functions are simpler and more flexible to use, e.g., lambda expressions can be subject to such a communication. As a drawback, the respective data is serialized (marshalled) in order to be sent and is de-serialized by the receiver. Serialization is omitted by the specialized functions, so using them for sending `int` and `float` values is almost always recommended.

5.4.2 C Interfacing for MPI

To give an impression of the C interfacing of OCamlMPI, consider the OCaml binding of the MPI function for a blocking send operation, which has a signature given by the following C declaration.

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Parameter `buf` points to a buffer containing `count` consecutive entries of type `datatype` to be sent. As the process invoking the function is defined as sender, the target process is given by its rank `dest` within a communicator `comm`. A communicator is a set of processes to which the scope of a communication can be restricted. To enable the receiver to identify a particular communication, a `tag` can be given.

The signature for the OCaml function to perform a blocking send is given by

```
val send_int: int -> rank -> tag -> communicator -> unit
```

For performing the conversions needed, a stub function connects the OCaml declaration to the call of the MPI function.

```
value caml_mpi_send_intarray(value data, value dest,
                             value tag, value comm)
{
  MPI_Send(&Field(data, 0), Wosize_val(data), MPI_LONG,
           Int_val(dest), Int_val(tag), Comm_val(comm));
  return Val_unit;
}
```

5.4.3 Installation and Usage of OCamlMPI

The installation of OCamlMPI for UNIX is straight forward. In order to build the respective OCaml module, three environment variables have to be set in the GNU makefile (The Free Software Foundation, 2005b):

- `DESTDIR` – the target installation directory to contain the module and interface files
- `MPIINCDIR` – the directory containing the MPI header files
- `MPILIBDIR` – the directory containing the MPI libraries

For initiating the compilation of the module and for installing the result in the target directory, make is called at the command line with the respective targets as options.

```
make all
make install
```

Unfortunately, the MPI standard does not define precisely how MPI source code is compiled or called. For some distributions it is sufficient to give the library as argument for the compiler. In this case, setting the above makefile variables appropriately is sufficient to build the OCaml binding. Other MPI implementations, like MP-MPICH, are shipped with dedicated front ends for C and Fortran compilers. In this case, the OCamlMPI makefile has to be modified:

- The compiler variable must be set to the respective compiler front end executable, like `CC=mpicc`.
- Each call of the OCaml compiler (`$(OCAMLC)` or `$(OCAMLOPT)`) needs an extra option giving the compiler front end `-cc $(CC)`.

After a successful installation, the binding can be used by specifying the installation path at the OCaml compiler command line (`-I`*directory*). Each binding provided by OCamlMPI has a brief description in the respective interface file `mpi.mli`.

5.4.4 OCamlMPI – the Starfish Implementation

An alternative implementation of MPI in OCaml is given by Agbaria and Friedman (1999), who implemented core functionality of the MPI 1 standard. This implementation, coincidently named OCamlMPI, too, is part of the Starfish environment, which aims at providing a fault-tolerant and dynamic execution of MPI programs on a cluster of workstations. Currently, the communication hardware supported is (high-speed) Ethernet, for convenience, and Myrinet, for performance.

Fault-tolerant MPI is especially an issue for Grid computing (Gabriel et al., 2003), which aims at coordinated resource sharing, combining computation resources dynamically depending on the needs of the application.

The architecture of Starfish consists of *application processes*, running MPI programs, and monitoring *Starfish daemons*. A group of cooperating daemons forms the parallel environment and is responsible for spawning application processes, asserting fault tolerance and managing the cluster setting. Especially for the latter the Starfish environment handles dynamic changes in the cluster configuration.

The subsequent example of the Karatsuba polynomial product was not developed for Starfish but for the MPI binding by Leroy (2001) described before. Unlike the Starfish implementation, its shallow interface uses only the specified C interface of MPI. Therefore, it can be used in combination with an arbitrary C implementation of MPI[1].

5.5 Parallel Meta-programming

5.5.1 A Three-tier Design

The three-tier design Herrmann (2005) uses for implementing a parallel Karatsuba polynomial product is illustrated in Figure 5.1 on the next page. The contents of each layer is chosen according to the respective domain. Each layer uses functionality of the respective layer below and provides its own functionality by an interface to the layer above.

1. The lowest layer implements a specification language for parallelism. This language is an embedded DSL composing parallel and sequential operations in order to describe a static communication structure. Guided by a cost model, this structure is analysed by a staged interpreter on each process for generating the respective process specific code.

2. The middle layer is a parallel implementation of a static computation structure. It uses the specification language of the lowest layer as an abstraction from the actual parallel environment as well as high-level features of the host language MetaOCaml, like recursion and higher-order functions.

[1] The cluster of workstations available for the benchmarks uses an SCI interconnection, which is not supported by the latest Starfish version 0.2.

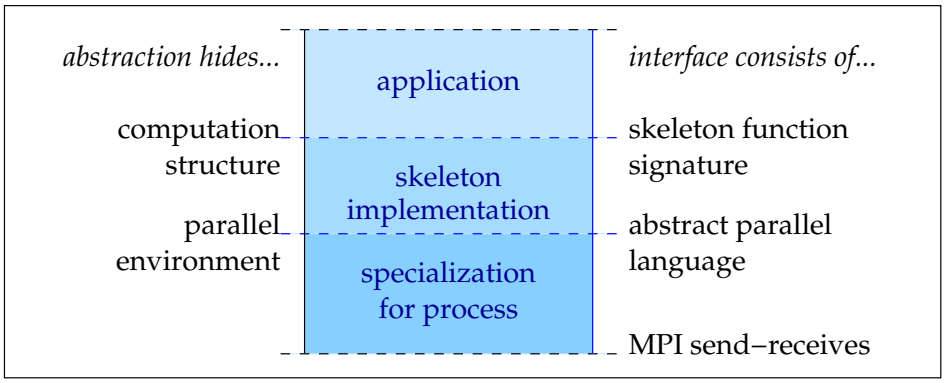


Figure 5.1: Three-tier architecture for implementing parallel applications.

3. The highest layer uses a skeleton interface of the computation structure in the layer below. The application implements computation fragments and provides them as arguments for the call of the skeleton.

5.5.2 Implementation of the Layers

Herrmann (2005) implemented the Karatsuba polynomial product in the highest layer, making use of a divide-and-conquer skeleton. At this layer, the skeleton appears as higher-order function `dc`. Application implementers who want to make use of the computational structure of this skeleton have to implement three functions to be passed as arguments: a function describing the basic case (`basic`), a function describing how to split the problem into subproblems (`divide`) and a function describing how to combine partial solutions to a common solution (`combine`). Furthermore, two `int` values specify the depth (`depth`) and number (`degree`) of recursive descents per step of the recursion. The skeleton function is used by calling

```
dc degree basic divide combine depth
```

The underlying implementation of the divide-and-conquer computation is invisible to the program part using the skeleton interface. Though it is implemented by a parallel specification in this example, it could as well be a sequential recursive descent.

Similar to Herrmann (2005), we give the parallel specification language as MetaOCaml data structure. This language is used to implement the the skeleton.

```
type 'a exp =
    Atom of ('a -> 'a)
  | Seq of (int * (int -> 'a exp))
  | Par of (int * (int array -> int -> 'a exp))
```

This embedded language concentrates exclusively on aspects of defining a static parallel task structure since MetaOCaml as host language provides all other features needed. Constructor `Atom` f declared a host language program f to be part of the

parallel program. `Seq (`n`,` e`)` sequences n steps of e, where e is parameterized with the current step index. `Par (`n`,` e`)` defines parallel execution of e, where e is parameterized with the communication context in terms of an array of partner identifiers, and its own task identifier.

As a consequence of the implementation of the divide-and-conquer skeleton function, the code is generated and executed in two steps, as the following example call in the interactive environment demonstrates.

```
# let parprog = dc 3 karat_basic karat_divide karat_combine 2;;
val parprog : ('_a, int array) code exp = Par (3, <fun>)
```

Constructor `Par` holds its sub-expressions as function, which is evaluated in a first step during the interpretation of the embedded language. As the type argument `('_a, int array) code` indicates, the result of the interpretation is staged code, which implies staged code in the host language fragments of the parallel program. As second step, the resulting code object is compiled with the MetaOCaml run operator and run on each process

The interpretation implemented in the lowest layer is a partial evaluation `peval` of the input program with respect to the communication context for a single process. For 9 processes, as Figure 5.2 on the facing page shows, the only work to be done by process 5 is to receive data from process 3, to perform the basic computation and to send back the result to complete the operation. Thus, `peval` generates code for process 5 which performs exactly these operations.[2]

```
 .<let y_3 =
    let y_2 =
3    begin
      x;
      let y_1 = receive 3 1 in
      divide 2 y_1
     end in
8  basic y_2 in
   send y_3 3 1;
   y_3>.
```

5.6 Further Optimizations

Preceding experiments by Herrmann (2005) revealed significant speedups. Nevertheless, the implementation still leaves many options of further optimization. Especially the computation-intensive parts of the basic solution are a lucrative target – they can be tackled independently from the parallel code.

The basic solution passed to the divide-and-conquer skeleton essentially consists of a sequential implementation of the application. In the case of the Karatsuba polynomial product, this solution consists of a sequential recursion. Because the overhead

[2] The pretty printer of MetaOCaml would also annotate `receive`, `send`, `basic` and `divide` as cross-stage persistent variables, i.e., variables which are defined in a lower stage than their use. These annotations have been removed in order to improve the readability.

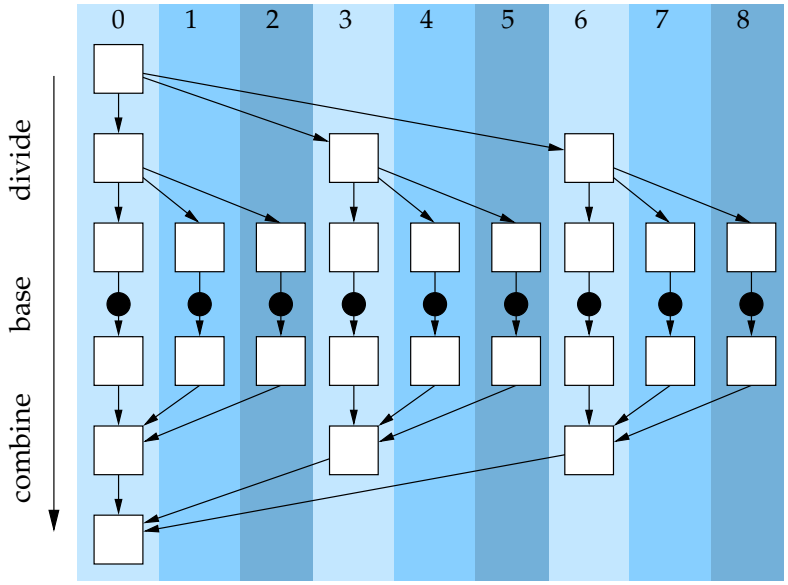


Figure 5.2: Divide-and-conquer with degree 3 and depth 2

of a recursive descent would dominate the computation for small operands, the recursive solution is switched to a sequential solution for input of a length below a given threshold.

```
  let rec karat_seq xs ys =
    let n = Array.length xs in
    if n<=16
    then                          (* solution for small problem sizes *)
      let zs = Array.make (2*n) 0 in
      let _ = karat_small_seq (n, xs,ys,zs) in
      zs
    else                          (* solution for large problem sizes *)
      let low   = karat_seq (lowpart    xs) (lowpart    ys)
      and high  = karat_seq (highpart   xs) (highpart   ys)
      and mixed = karat_seq (mixedparts xs) (mixedparts ys) in
      let zs = Array.append low high in
      for i=0 to n-1 do
        zs.(i+n/2) <- zs.(i+n/2) + mixed.(i) - low.(i)-high.(i)
      done;
      zs
```

The original sequential implementation by Herrmann (2005) is lambda lifted to a new function `karat_small_seq`, which makes the local bindings explicit as function parameters.

```
  let karat_small_seq = begin fun (n,xs,ys,zs) ->
      for i=0 to n-1 do
        for j=0 to n-1 do
4         zs.(i+j) <- zs.(i+j) + xs.(i) * ys.(j)
        done
      done; 0
  end
```

This function offers a very promising optimization by offshoring due to the following reasons:

- It is repeatedly called as base case of the Karatsuba polynomial product. In fact, the major portion of the computational work is done in this function.
- The MetaOCaml constructs used in this function are elements of the language restrictions imposed by the offshoring mechanism: each of them has a corresponding construct in the target language C.
- The lambda lifted form reduces the overhead of offshoring to a minimum because compilation and linking has to be done once only at the beginning of the execution.

The offshoring optimization requires minimal changes to the code.

```
  let karat_small_seq_code = .< begin fun (n,xs,ys,zs) ->
      for i=0 to n-1 do
3       for j=0 to n-1 do
          zs.(i+j) <- zs.(i+j) + xs.(i) * ys.(j)
        done
      done; 0
  end >.
8
  let karat_small_seq = .!{Trx.run_gcc} karat_small_seq_code
```

Now, the function is surrounded with code brackets and run with the respective offshoring run operator.

5.7 Experimental Results

As for all benchmarks before, the experiments for Karatsuba were run on a cluster of 1 GHz Dual Pentium III machines with 512 MB of memory running Fedora Core 1 Linux. The nodes of this cluster are linked with an SCI interconnect. The MPI implementation used was MP-MPICH, which provides both an implementation of the C library and a run-time environment for running compilations on the cluster.

The parallel implementation was compiled with two different versions of MetaOCaml: For both the plain bytecode compilation and the version using offshoring the latest MetaOCaml release available was used (3.08 alpha 023). For the native code compilation an unofficial development version based on OCaml 3.07 was used because the latest native code compiler revealed bugs in the allocation of arrays.

compilation	#procs	16	17	18	19
bytecode	1	126,91	382,45	1154,02	3493,54
	3	42,59	127,89	384,86	1164,29
	9	14,56	43,28	129,39	389,58
native	1	10,16	30,95	97,71	326,82
	3	3,72	10,56	32,77	105,59
	9	1,62	4,11	11,58	35,92
bytecode	1	50,45	152,46	462,84	1416,76
+ offshoring	3	17,09	51,32	154,86	471,15
	9	6,04	17,71	53,01	159,17

Table 5.2: Benchmark results on Fedora Core 1, Dual PIII, 512 MB, SCI interconnect.

Timings were done for operand arrays of sizes from 2^{16} to 2^{19} and process numbers of 1, 3 and 9.The powers of 3 result from the division degree 3 of the Karatsuba algorithm. Though the use of both processors on each node of the cluster would have reduced the need for communication, each participating process had to be run on a separate node because of faults in the parallel runtime environment.

Though the absolute results shown in Table 5.2 do not resemble directly the experiments made by Herrmann (2005), the relative speedups are more or less the same. Additionally, native execution shows enormous speedups of more than one order of magnitude compared to bytecode execution. The offshoring optimization, which required minimal changes of the original code, is almost three times faster than the corresponding bytecode run. Unfortunately, the combination of offshoring and native compilation resulted in run-time failures in the shared memory interface of MP-MPICH.

Chapter 6

Development Environment

6.1 The MetaOCaml Distribution

Current releases of the MetaOCaml distribution are branches of the official Objective Caml releases. To refer to the respective release, each MetaOCaml release adds its number to that of the underlying OCaml version. E.g., the current release of MetaOCaml 3.08 alpha 023 is based on OCaml 3.08.

OCaml

The language implementation provided by the basic OCaml installation consists of compilers for generating machine-independent bytecode (`ocamlc`) and native code for widespread platforms (`ocamlopt`). A standard library complements the core library of OCaml. Modules of these libraries are available in the run-time environment of executables. Additional libraries provide implementations of selected aspects, like access to Unix system routines or basic graphics routines.

The development environment of OCaml provides a number of useful tools helping to gear up the development process:

- A top-level system (`ocaml`) providing a loop-style interactive session. Code phrases given at the prompt are parsed, type-checked and compiled by the system. After a successful execution, the corresponding data type and returned value are printed on the display.
- Lexer and Parser generators (`ocamllex` and `ocamlyacc`). The design and usage of these tools are closely related to the *lex* and *yacc* commands known from many C environments.
- A debugger (`ocamldebug`) and a profiler (`ocamlprof`).
- An interfacing mechanism for calling external C functions.
- A syntax-aware pre-processor (`camlp4`).

MetaOCaml

As MetaOCaml introduces only few new constructs to the OCaml language, the changes to the OCaml distribution seem to be minimal from a user's point of view. Still, internal adaptations affect many parts of the system. The following are the most prominent of them:

- The scanner and the parser were extended to support the new language extensions.
- The type checker was adapted to support the extended type system using a type constructor to distinguish explicit code from internal values.
- A new top-level environment provides run-time code generation.
- A new compiler front end (`metaocamlc`) uses the new top-level environment.
- The interactive top-level environment provides additional pretty prints of code objects.

Also part of the MetaOCaml distribution are a native code compiler and the offshoring mechanism.

The interactive environment, which can be started with the command `metaocaml`, provides a good development testbed. Started with the command line argument `-dinstr`, the execution of a phrase produces an extra print-out of the generated bytecode. A simple MetaOCaml phrase defining a lambda expression in the second stage is compiled to the following code.

```
# let c = .< fun () -> 42 >. in .! c ;;
```

```
        const
          [0:
           [3:
4           "" 0a
            [0:
             [7:
              [0:
               [4: [0: "()"] 0a 0a] [0: [0: "" 1 0 11] [0: "" 1 0 23] 0a]
9              [0: CSP_value]]
              [0:
               [1: [0: 42]] [0: [0: "" 1 0 21] [0: "" 1 0 23] 0a]
               [0: CSP_value]]]
             0a]]
14         [0: [0: "" 1 0 11] [0: "" 1 0 23] 0a] [0: CSP_value]]
        push
        acc 0
        push
        getglobal Toploop!
19      getfield 34
        appterm 1, 3
```

Without going much into the details, we can see that the lambda expression is represented by a constant data structure (lines 1–15) instead of being translated to bytecode

instructions. As the code for the `let` binding was omitted by the compiler, this data structure is pushed directly onto the operand stack of the Caml virtual machine (line 16). In lines 18–20 the run operator is retrieved from the global environment and applied to the code object on the stack.

The application of the run operator transforms the code object to the following bytecode, which comprises the creation of a closure for the function code fragment at label `L1`. This code fragment is then linked dynamically to the running program.

```
        closure L1, 0
        return 1
L1:     const 42
        return 1
```

From version 3.08 alpha 023 on MetaOCaml supports native code compilation for i386 machines. The new command, `metaocamlopt`, provides a front end with the same look and feel as the OCaml native compiler. The development of this compiler is still not as mature as that of the bytecode compiler. The creation, compilation and dynamic linking of staged code is highly dependent on the underlying hardware and software architecture. As `ocamlopt` does not support dynamic linking, OCaml had to be patched (Karpov, 2005) providing the generation and dynamic loading of position independent OCaml objects.

6.2 Interfacing to C

The OCaml system provides a straight-forward facility for interfacing OCaml with C. This interface works in both directions, i.e., C modules may call OCaml functions and vice versa. A set of C macros supports the handling of the C representations of OCaml data objects. Since interfacing to C is described in-depth in the respective documentation by Leroy et al. (2004), this section gives only an introductory overview using the example of a power function.

6.2.1 Calling C from OCaml

Commonly, the interface to C consists of three parts, which have to be provided by the implementer.

A declaration to introduce an external function into the scope of the OCaml program.

A C stub function to mediate between OCaml and C, i.e., to convert OCaml values to C values and vice versa.

A C implementation providing the actual functionality.

External functions are declared in the implementation section of an OCaml program, i.e., an `.ml` file or a `struct...end` block. This declaration consists of the keyword `external`, the function name to be used in OCaml, the OCaml data type of this

function and the name of the stub function. The following line specifies an association of the function `power` to the stub function `power_stub`.

```
external power : float -> int -> float = "power_stub"
```

Optionally, this declaration can also be put into an interface section. This makes the implementation of `power` as a C function visible to clients of the module and enables them to inline the external call into their code.

An adequate implementation of the stub function can be as follows.

```
#include<caml/mlvalues.h>
#include<caml/memory.h>
...
value power_stub(value b, value e)
{
  CAMLparam2(b,e);
  CAMLreturn(copy_double( power(Double_val(b), Long_val(e))));
}
```

The name of this function, `power_stub`, is the one stated in the declaration, and the number of arguments is equal to the number of arrows in the corresponding OCaml type.

On the C side, all OCaml values are represented as objects of type `value`. All arguments and the return value of the stub function are of this type. Therefore, the principle task of a stub function should be the adequate conversion of these arguments to C data types: passing them to a C function which performs the actual operation (`power()` in our example) and creating a `value` object from the result of the operation. The included header file `mlvalues.h` provides macros to be used for these transformations.

Creating int values with the macros `Val_long()` or `Val_int()` is as simple as reading them with `Long_val()` or `Int_val()` because integers are represented as unboxed values. All other values, like arrays, closures or variant types, are boxed values, i.e., they consist merely of a pointer referencing a block on the run-time heap or stack. There are special macros for accessing these blocks and for creating new ones, like `copy_double()` used in the example.

To make the garbage collector aware of the values given as parameters, there are macros `CAMLparam`n`()` for each number n of parameters from 0 to 5 and a macro `CAMLxparam` for each additional parameter. Also, as replacement for the C keyword `return`, one should use the macro `CAMLreturn` (or `CAMLreturn0` for a function without a return value).

Special care is needed for functions with a number $n > 5$ of parameters. While the native compiler expects the stub function to receive n arguments of type `value`, the bytecode interpreter needs the stub function to have two parameters: a C array of type `value*` and an `int` argument specifying the arity n. Both types of stubs have to be implemented and noted in the respective declaration on the OCaml side (details are given by Leroy et al. (2004)).

Having implemented a stub function, the C function implementing the power function need not be aware of the fact that it is called from OCaml.

```
double power(double b, int e) {
  double acc = 1.0;
  while (e > 0)
  {
    acc *= b; e--;
  }
  return acc;
}
```

The approach of separating the actual operation from the type conversions is especially favorable if a C implementation exists in advance, e.g., as a third-party library.

6.2.2 Calling OCaml from C

The most common way of calling OCaml functions from C is by using callbacks. Callback macros are defined in the header file `caml/callback.h` and provide the mechanism to execute a closure given as an OCaml `value`. These macros are named `callback(f)`, `callback2(f,a)`, `callback3(f,a,b)`, etc., depending on the arity of the function represented by closure `f`.

OCaml functions given as parameters to the C function can be called directly using these functions. If a function is not given as parameter but only in the top-level environment of the calling OCaml program, its closure has to be obtained using a simple registering mechanism. To give an example, consider the following code fragments, which displays a C stub function `power_callback_stub` calling an OCaml function `power`. First, the stub function is made available by the respective `external` declaration in OCaml.

```
external power_stub: float -> int -> float = "power_callback_stub"
```

The stub function is now known to the OCaml program by the name `power_stub`. Then, the OCaml function to be called from the C function is registered by using the respective function from the `Callback` module.

```
Callback.register "power" power
```

To be used in C, the closure of function `power` must be retrieved by calling function `caml_named_value()`. It takes the same label as argument `"power"` which was given to the register function on the OCaml side.

```
value power_callback_stub(value b, value e)
{
  static value *closure_f = NULL;
  CAMLparam2(b, e);

  if (closure_f == NULL)
  {
    closure_f = caml_named_value("power"); /* retrieval of closure */
  }

  CAMLreturn (callback2(*closure_f, b, e)); /* call of closure */
}
```

Note the solution to look up the function power only once, at the first time of its usage. The keyword `static` makes the variable pointing to this function survive the exit and re-entry of the stub function.

6.2.3 Linking to OCaml Code

For the use of external C libraries with OCaml programs, there are two principle options provided by the compiler front end: static or dynamic linking. For plain OCaml programs, the OCaml compiler `ocamlc` produces executables which consist solely of the object files and rely on the standard run-time environment providing the core functionality. In order to use static linking of external C code, a custom run-time environment has to be built using the command line argument `-custom`. In custom mode, the compiler scans the object files for the required primitives and builds a suitable run-time system. Given an external module `c_module.c` and an OCaml module `main.ml` using this C module, the following steps would have to be carried out for building a stand-alone executable.

```
gcc -I $OCAML/lib/ocaml -c c_module.c
ocamlc -c main.ml
ocamlc -o main -custom c_module.o main.cmo
```

If `ocamlopt` is used for the compilation, argument `-custom` is not needed at the command line because the native code compiler always builds a custom run-time environment.

For dynamic linking, the respective C code has to be compiled as dynamic C module (`.so` in UNIX, `.dll` in Windows). For a dynamic C module `dllmylib.so`, e.g., arguments `-dlib -lmylib` provided at the command line of `ocamlc` would yield an executable using dynamic linking.

Chapter 7

Conclusions and Perspectives

Looking at the case studies in this thesis, MetaOCaml shows a promising potential for an application in the domain of high-performance computing. On the one hand, a source of the significant speed gains lies in the core implementation of OCaml, which generates faster executables than many other compilers of functional languages. On the other hand, the language extensions introduced by MetaOCaml offer multi-staged programming in OCaml, which allows high-level code optimizations at run time.

We demonstrated the combination of MetaOCaml with different tools and techniques, which allow optimizations that are far more time-consuming, difficult and error-prone to implement in pure C or Fortran. Table 7.1 provides a summary of these techniques.

General Optimization Techniques

The basis of all optimizations is knowledge about which software tools to choose and how they influence the running time of an executable. This concerns issues like the tuning of the run-time system or guidelines for speed-aware coding. Especially floating-point operations and values need extra care in order to enable OCaml to use an alternative unboxed implementation. Unboxed floating-point operations make OCaml a recommendable choice for numerical computing. Additionally, the efficient implementation of function applications in OCaml allows the use of recursion without heavy speed losses.

Using the few new constructs introduced by MetaOCaml, the implementer can generate optimized code while being guided by a type system which guarantees the type correctness of generated code statically. Furthermore, code is generated at run time, which allows the application of optimizations depending on run-time information.

The current implementation of MetaOCaml provides bytecode and native compilation for i386 platforms. Nevertheless, the native-code compiler, which is more relevant for high-performance computing, is still in an unstable development state.

OCaml, as basis of the MetaOCaml distribution, provides a simple interfacing mechanism to access C primitives like OCaml function. This allows to integrate hand-optimized C code or external C libraries into (Meta)OCaml programs. This mechanism

Optimization	Trade-off	Recommendation	Section
OCaml instead of C.	abstraction vs. low-level optimization	Integrate optimized C fragments with the interfacing mechanism.	6.2
Tuning of garbage collection.	speed vs. memory	Recommended. In most cases lack of memory is not a problem.	2.1
Deactivation of loop boundary checks.	speed vs. safety	Only if high-level optimizations are not sufficient.	2.2.1
Preference of monomorphism.	speed vs. abstraction	Avoid polymorphism for float arrays, float expressions and functions using float values.	2.2.2, 3.1
Currying instead of tupling.	speed vs notation	Try to avoid the passing of tuples.	2.3.2
Loops or recursion?	iterative vs. recursive problem	Loops for iterative algorithms, especially for native-code execution. Recursion for traversing recursive structures.	2.3.1
MetaOCaml instead of C.	high-level vs. low-level optimization	Reduces the optimization effort. Complements many other optimization techniques.	3
Loop unrolling.	speed vs. memory	Recommended. Provides run-time customization.	2.4, 3.4.1, 4
Code generation from abstract specification.	speed vs. effort	Recommended. The effort is limited, the result is significant.	4, 5.5
Code generation for message passing.	speed vs. effort	Clean run-time optimization for parallel environment.	5.5
Offshoring instead of explicit interfacing to C.	hygienic coding vs. expressiveness	Recommended whenever expressive enough. Best optimization for computation intensive loop nests.	3.4.2, 5.6
Interfacing to C instead of OCaml.	speed vs. type safety	Recommended for optimizations not possible in plain OCaml.	6.2
JIT instead of native compilation.	platform independence vs. speed	Not recommended for HPC due to low speedups. Platform independence is not a big issue in HPC.	2.6

Table 7.1: List of all optimizations presented.

holds the implementer accountable for type-correct handling of OCaml values in C code. Still, for high-performance computing, interfacing to C is a good choice in order to benefit from low-level, machine-dependent optimizations hard to reproduce in MetaOCaml.

Implicitly heterogeneous multi-stage programming provides a restricted but type safe means of exporting OCaml computations to C or Fortran. Code fragments consisting of constructs which have a counterpart in C or Fortran are transformed to the respective target language by a lightweight translation process, called *offshoring*. The source code is then compiled by an optimizing C or Fortran compiler and linked to the running OCaml program. As this process is implicit, offshoring is hygienic in the sense that the application programmer has no need to leave the host language. At the same time, he benefits from all optimizations applied by the external compiler. The offshoring mechanism is part of the official MetaOCaml release but is still in a state of infancy. Nevertheless, it provided the best results in terms of speedup, abstraction and type safety.

The current implementation of just-in-time compilation for OCaml bytecode turned out to be a bad choice for high-performance computing. Though bytecode programs run significantly faster, they cannot compete with a native compilation.

Matrix Multiplication as Testbed for Optimization Techniques

We presented three case studies, each demonstrating a certain aspect in order to value MetaOCaml as tool for program optimizations.

The example matrix multiplication implementations show that C allows low-level optimizations which cannot be applied directly in OCaml without modifying the compiler. Especially the use of a cache-friendly memory allocation made an iterative matrix multiplication in C run more than three times faster than the respective implementation in OCaml.

Still, MetaOCaml simplifies the implementation of high-level optimizations which would be complex and error-prone to reproduce in C. For the multiplication of $n \times n$ matrices, the fastest program ran about 2.5 times faster than the fastest implementation of an iterative matrix multiplication in C. This OCaml implementation uses a variation of Strassen's (1969) algorithm, which switches to an offshored iteration for subproblems below a certain threshold. As offshoring provides code generation at run time, an optimal threshold could be approximated by preceding minor benchmarks. Furthermore, this implementation uses exclusively the high-level language of MetaOCaml and, still, benefits from the implicit low-level optimizations performed by the C compiler. Therefore, offshoring is highly recommended for all situations where the restricted syntax of the object language is expressive enough. Unfortunately, the current official release of offshoring is rather unstable and still needs time to reach a certain level of maturity.

MetaOCaml turned out to be a good choice for implementing a loop unrolling. By using run-time code generation it can be adjusted to perform well with the cache size of the target machine. Respective benchmarks of a Strassen implementation were among

the best of all results. Unfortunately, the latest native-code implementation of MetaOCaml still is in a state of infancy as tests revealed a defective allocation of arrays.

A rather simple optimization in MetaOCaml is the deactivation of boundary checks in array indexing operations. One of our implementations using arrays of arrays profited by this optimization showing a speedup of 2.68. Still, the implementer should be aware of the risks of using these unsafe operations, which may obscure invalid array accesses.

Domain-specific Optimizations

Another aspect of multi-staged programming is presented in a case study on image processing. Using MetaOCaml for the implementation of a domain-specific language can simplify the development of an optimizing compiler back end. After scanning and parsing an image processing specification, an online partial evaluater constructs a simplified AST. The subsequent code generation phase combines second stage code fragments to produce an optimized code object. This residual function is then applied on each pixel of an input image in order to produce a processed output image. Compared to an implementation using pure interpretation, our optimizations pay off already for input images of the size of GUI icons.

For parallel programming, the combination of the Message Passing Interface (MPI) with MetaOCaml allows separate optimizations within each processes of a parallel run-time environment. In this thesis, we also presented a design by Herrmann (2005), which has three layers of abstraction: in the middle layer, an embedded parallel specification language is used to implement the computational structure of a skeleton function. In the layer above, an application developer can use such a skeleton without the need for expert knowledge in the domain of parallel computing. In the layer below, parallel specifications are specialized for each target process. This is accomplished by a function which generates code at run time, depending on the process identifier in a distributed parallel environment.

Complementing the parallelization with MPI, the offshoring of a small but computation intensive fragment of code can generate additional speedup. In the benchmark shown, this optimization produced a speedup of more than 2 for the implementation of a parallel Karatsuba multiplication.

Final Conclusions

To put it in a nutshell, we showed that MetaOCaml can be combined with different tools and techniques to write optimizations at any implementation level. Each solution can be tailored to the needs of the respective application domain. Though some of the tools we used were still rather unstable, we were able to narrow significantly the gap between efficient but error-prone, hand-optimized C code and high-level software design using functional and object oriented abstractions.

Appendix A

Matrix Multiplication Benchmark Results

- Architecture: 1 GHz Dual Pentium III, 512 MB memory.
- Operating System: Fedora Core 1 Linux.
- GCC 3.3.2 for all C programs and external C modules
- Native compiler of
  - MetaOCaml version 3.07+2[+Shared] for all programs not using offshoring.
  - MetaOCaml version 3.08.0 alpha 023 for all programs using offshoring.
- Configuration:
  - Optimization option `-O3` for all C implementations.
  - Tile width 128 ($= 2^7$) for recursive implementations.
  - Unrolling width 32 ($= 2^5$) for all `+U` implementations.
  - Boundary checks disabled for all implementations.[1]

The identifiers used for the implementations are given in Table 3.1 on page 36 and Table 3.2 on page 52. The entries of column n give the logarithmic input matrix width of each respective run. An asterisk ($*$) marks the fastest runs for both $n = 10$ and $n = 11$.

[1] In the case of offshoring, the compiler option for unsafe array operations makes the executable fail at run-time. Apparently, the implementation of offshoring coming with MetaOCaml 3.08 alpha 023 does not generate code for functions `Array.unsafe_get` and `Array.unsafe_set`. Nonetheless, ordinary array accesses, which have a bound-check semantics in OCaml, are mapped directly to unsafe array accesses in C.

Implementation		n	pre	proc	post	total
IterOrd	OAa	7	0,000	0,039	0,000	0,039
IterOrd	OAa	8	0,000	0,869	0,000	0,869
IterOrd	OAa	9	0,000	10,781	0,000	10,781
IterOrd	OAa	10	0,000	98,990	0,000	98,990
IterOrd	OAa	11	0,000	800,102	0,000	800,102
IterOrd	CAa	7	0,002	0,020	0,001	0,022
IterOrd	CAa	8	0,008	0,377	0,003	0,388
IterOrd	CAa	9	0,031	3,518	0,013	3,562
IterOrd	CAa	10	0,114	29,074	0,049	29,238
IterOrd	CAa	11	0,423	363,876	0,208	364,507
IterRow	OA	7	0,004	0,064	0,002	0,069
IterRow	OA	8	0,014	2,605	0,006	2,625
IterRow	OA	9	0,047	24,335	0,025	24,407
IterRow	OA	10	0,185	194,167	0,102	194,454
IterRow	OA	11	*RangeLimitExceeded*			
IterRow	OB1	7	0,002	0,210	0,001	0,213
IterRow	OB1	8	0,008	3,108	0,007	3,123
IterRow	OB1	9	0,034	28,341	0,022	28,398
IterRow	OB1	10	0,132	225,029	0,099	225,259
IterRow	OB1	11	0,524	1812,737	0,444	1813,704
IterRow	CA	7	0,000	0,025	0,000	0,025
IterRow	CA	8	0,000	0,796	0,000	0,796
IterRow	CA	9	0,000	8,533	0,000	8,533
IterRow	CA	10	0,000	74,277	0,000	74,277
IterRow	CA	11	0,000	613,138	0,000	613,138
Iter2D	OB2	7	0,002	0,257	0,001	0,261
Iter2D	OB2	8	0,010	2,856	0,008	2,874
Iter2D	OB2	9	0,041	27,763	0,026	27,830
Iter2D	OB2	10	0,157	221,194	0,104	221,455
Iter2D	OB2	11	0,620	1791,628	0,476	1792,724
Iter2D	OB2+C	7	0,002	0,037	0,001	0,041
Iter2D	OB2+C	8	0,010	0,796	0,007	0,813
Iter2D	OB2+C	9	0,038	9,209	0,025	9,273
Iter2D	OB2+C	10	0,155	74,193	0,106	74,454
Iter2D	OB2+C	11	0,620	588,413	0,480	589,513
IterTrp	OAa	7	0,003	0,037	0,000	0,040
IterTrp	OAa	8	0,008	0,700	0,000	0,709
IterTrp	OAa	9	0,053	5,576	0,000	5,629
IterTrp	OAa	10	0,262	44,436	0,000	44,698
IterTrp	OAa	11	1,167	350,046	0,000	351,212
IterTrp	OB2	7	0,004	0,085	0,003	0,093
IterTrp	OB2	8	0,022	1,073	0,013	1,108
IterTrp	OB2	9	0,095	8,374	0,033	8,502

IterTrp	OB2	10	0,383	67,368	0,282	68,034
IterTrp	OB2	11	1,540	529,940	0,653	532,133
IterTrp	CAa	7	0,002	0,018	0,001	0,021
IterTrp	CAa	8	0,009	0,528	0,003	0,540
IterTrp	CAa	9	0,042	4,203	0,012	4,256
IterTrp	CAa	10	0,163	33,711	0,050	33,923
IterTrp	CAa	11	0,696	268,958	0,205	269,859
IterTrp	CA	7	0,001	0,019	0,000	0,020
IterTrp	CA	8	0,005	0,547	0,000	0,551
IterTrp	CA	9	0,023	4,357	0,000	4,380
IterTrp	CA	10	0,097	34,912	0,000	35,009
IterTrp	CA	11	0,453	278,716	0,000	279,169
IterTrp	OAa+U	7	1,017	0,021	0,000	1,037
IterTrp	OAa+U	8	0,764	0,572	0,000	1,336
IterTrp	OAa+U	9	0,833	4,559	0,000	5,392
IterTrp	OAa+U	10	1,001	36,289	0,000	37,290
IterTrp	OAa+U	11	2,240	286,750	0,000	288,990
IterTrp	OB2+C	7	0,002	0,033	0,001	0,037
IterTrp	OB2+C	8	0,010	0,548	0,007	0,565
IterTrp	OB2+C	9	0,039	4,377	0,024	4,439
IterTrp	OB2+C	10	0,154	34,991	0,108	35,252
IterTrp	OB2+C	11	0,625	279,111	0,482	280,219
IterTrp	OAa+O	7	0,873	0,035	0,000	0,908
IterTrp	OAa+O	8	0,809	0,588	0,000	1,396
IterTrp	OAa+O	9	0,865	4,750	0,000	5,615
IterTrp	OAa+O	10	1,124	37,459	0,000	38,583
IterTrp	OAa+O	11	2,314	296,909	0,000	299,223
RecOrd	OA	7	0,006	0,064	0,003	0,073
RecOrd	OA	8	0,031	0,499	0,014	0,544
RecOrd	OA	9	0,117	3,935	0,050	4,102
RecOrd	OA	10	0,512	31,449	0,211	32,172
RecOrd	OA	11	*RangeLimitExceeded*			
RecOrd	OB1	7	0,004	0,109	0,004	0,118
RecOrd	OB1	8	0,026	1,073	0,014	1,113
RecOrd	OB1	9	0,110	7,037	0,049	7,197
RecOrd	OB1	10	0,490	55,228	0,223	55,941
RecOrd	OB1	11	2,339	443,985	1,008	447,332
RecStr	OA	7	0,006	0,041	0,003	0,050
RecStr	OA	8	0,029	0,317	0,012	0,358
RecStr	OA	9	0,122	2,523	0,051	2,696
RecStr	OA	10	0,526	16,945	0,219	17,690
RecStr	OA	11	*RangeLimitExceeded*			
RecStr	OB1	7	0,004	0,090	0,004	0,098
RecStr	OB1	8	0,024	0,755	0,013	0,792
RecStr	OB1	9	0,110	5,373	0,049	5,532

RecStr	OB1	10	0,499	39,045	0,231	39,776	
RecStr	OB1	11	2,304	320,361	15,698	338,363	
RecStr	OA+U	7	1,116	0,045	0,007	1,168	
RecStr	OA+U	8	0,816	0,259	0,017	1,091	
RecStr	OA+U	9	1,186	1,737	0,049	2,972	
RecStr	OA+U	10	1,382	12,135	0,201	13,718	
RecStr	OA+U	11	*RangeLimitExceeded*				
RecStr	OB1+C	7	0,004	0,020	0,004	0,028	
RecStr	OB1+C	8	0,026	0,226	0,013	0,264	
RecStr	OB1+C	9	0,110	1,892	0,049	2,051	
RecStr	OB1+C	10	0,575	15,098	0,240	15,913	
RecStr	OB1+C	11	2,359	150,010	14,237	166,606	*
RecStr	OA+O	7	0,731	0,024	0,006	0,760	
RecStr	OA+O	8	0,753	0,207	0,012	0,973	
RecStr	OA+O	9	1,109	1,479	0,054	2,642	
RecStr	OA+O	10	1,250	10,728	0,217	12,194	*
RecStr	OA+O	11	*RangeLimitExceeded*				

Bibliography

Adnan M. Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *Proceedings of the IEEE Symposium on High Performance Distributed Computing*, pages 167–176, 1999.

Gene Myron Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, Vol. 30, pages 483–485. AFIPS Press, Reston, Va., 1967.

Greg Burns, Raja Daoud and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of the Supercomputing Symposium*, pages 379–386, 1994. `http://www.lam-mpi.org/`.

Cristiano Calcagno, Eugenio Moggi and Walid Taha. ML-like inference for classifiers. In *Proceedings of the European Symposium on Programming (ESOP'04)*, Lecture Notes in Computer Science 2986, pages 79–93. Springer-Verlag, 2004.

The Caml Consortium. The Caml language, 2005. `http://caml.inria.fr/`.

Emmanuel Chailloux, Pascal Manoury and Bruno Pagano. *Développement d'Application avec Objective CAML*. O'Reilly France, 2000. Preliminary English translation: `http://caml.inria.fr/oreilly-book/`.

Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala and Mithuna Thottethodi. Recursive array layouts and fast parallel matrix multiplication. *IEEE Transaction on Parallel and Distributed Systems*, 13(11):1105–1123, November 2002.

Albert Cohen, Sébastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann and David Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 2005. Accepted for publication.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz and Walid Taha. DSL implementation in MetaOCaml, Template Haskell and C++. In Christian Lengauer, Don Batory, Charles Consel and Martin Odersky, editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science 3016, pages 51–72. Springer-Verlag, 2004.

Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1–2):243–287, 2000.

Jason Eckhardt, Roumen Kaiabachev, Emir Pašalić, Kedar Swadi and Walid Taha. Implicitly heterogeneous multi-staged programming. In *Proceedings of the Fifth Conference on Generative Programming and Component Engineering (GPCE'05)*, 2005. `http://www.cs.rice.edu/~taha/publications.html`.

Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

E. Gabriel, G. Fagg, A. Bukovsky, T. Angskun and J. Dongarra. A fault–tolerant communication library for grid environments. In *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS'03), International Workshop on Grid Computing*, 2003.

Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004.

William Gropp, Ewing Lusk, Nathan Doss and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996. `http://www-unix.mcs.anl.gov/mpi/mpich/`.

Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs and Koen G. Langendoen. *Modern Compiler Design*. John Wiley & Sons, 2000.

John L. Gustafson. Reevaluating Amdahl's law. *Comm. ACM*, 31(5):532–533, 1988.

Christoph Herrmann and Tobias Langhammer. Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Science of Computer Programming*, 2005. To Appear.

Christoph A. Herrmann. Generating message-passing programs from abstract specifications by partial evaluation. *Parallel Processing Letters*, 15(3):305–320, September 2005.

Karel Hrbacek and Thomas Jech. *Introduction to Set Theory*. Marcel Dekker, Inc., 3rd edition, 1999.

Intel® Compilers. Intel® C++ Compiler, 2005. `http://www.intel.com/software/products/compilers/`.

ISO/IEC JTC1/SC22/WG14. ISO/IEC 9899:1999, 1999. `http://www.open-std.org/JTC1/SC22/WG14/`.

Neil Jones, Carsten K. Gomard and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. `http://www.dina.dk/~sestoft/pebook/pebook.html`.

A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 12:293–294, 1962. In Russian.

Vassili Karpov. SCaml – An OCaml patch for dynamic loading of shared objects, 2005. `http://www.boblycat.org/~malc/scaml/`.

Christian Lengauer. *Program Optimization in the Domain of High-Performance Parallelism*, pages 73–91. Lecture Notes in Computer Science 3016. Springer-Verlag, 2004.

Xavier Leroy. OCamlMPI – an interface with the MPI message-passing interface, 2001. `http://pauillac.inria.fr/~xleroy/software.html`.

Xavier Leroy. Objective Caml – questions and answers – making code run fast, 2002. `http://caml.inria.fr/pub/old_caml_site/ocaml/speed.html`.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy and Jérôme Vouillon. *The Objective Caml system release 3.08, Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

Stanley B. Lippman and Josée Lajoie. *C++ Primer*. Addison-Wesley, 1998.

MPI Forum. *MPI-2: Extensions to the Message Passing Interface*. University of Tennessee, Knoxville, Tennessee, July 1997. `http://www.mpi-forum.org`.

Open Management Group. *UML 2.0 Superstructure Specification*. Open Management Group, Inc. `http://www.omg.org/technology/documents/formal/uml.htm`.

Martin Pöppe, Karsten Scholtyssik, Silke Schuch, Joachim Worringen Rainer Finocchiaro and Carsten Clauss. MP-MPICH – User Documentation & Technical Guide, 2005. `http://www.lfbs.rwth-aachen.de/`.

William H. Press, Brian P. Flannery, Saul A. Teukolsky and William T. Vettering. *Numerical Recipes in Pascal*. The Art of Scientific Computing. Cambridge University Press, 1989.

Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01)*, Lecture Notes in Computer Science 2196, pages 2–44. Springer-Verlag, 2001.

Basile Starynkevitch. OCamlJIT – a faster just-in-time OCaml implementation. Unpublished, 2004. `http://cristal.inria.fr/~starynke/ocamljit.html`.

Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

Walid Taha. A gentle introduction to multi-stage programming. In Christian Lengauer, Don Batory, Charles Consel and Martin Odersky, editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science 3016, pages 30–50. Springer-Verlag, 2004.

Walid Taha, Cristiano Calcagno, Xavier Leroy, Ed Pizzi, Emir Pasalic, Jason Lee Eckhardt, Roumen Kaiabachev and Oleg Kiselyov. MetaOCaml – A compiled, type-safe, multi-stage programming language, 2005. `http://www.metaocaml.org/`.

The Free Software Foundation. GNU Lightning library, 2005/a. `http://www.gnu.org/software/lightning/`.

The Free Software Foundation. GNU project, 2005/b. `http://www.gnu.org/software`.

The Open Group. ISO/IEC 9945:2003, 2003. Including POSIX.2., `http://www.unix.org/version3`).

Jeyarajan Thiyagalingam, Olav Beckmann and Paul H. J. Kelly. Improving the performance of morton layout by array alignment and loop unrolling — reducing the price of naivety. In Lawrence Rauchwerger, editor, *LCPC 2003: Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 2958, pages 241–257. Springer Verlag, October 2003.

The links to the web pages are of September 1, 2005.

Index

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich diese Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe, dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind und dass diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, 6. September 2005

Tobias Langhammer