Hybrid Chunking
Department of Computer Science
Master of Science

Sven Auhagen

March 26th, 2012
Abstract

Manticore is a compiler for the parallel programming language Parallel ML (PML), which is based on Standard ML. PML supports parallelism using an implicitly-threaded model where the programmer annotates parts of the program that should be run in parallel. For example, in a recursive divide-and-conquer algorithm, the programmer could use a parallel tuple annotation to specify that recursive calls should be done in parallel. The Manticore compiler replaces the parallel tuples with futures that encapsulate the potential parallel work. At runtime, work stealing is used to balance the parallel workload.

For many programs, this approach has the problem that it introduces too much fine-grained parallelism, which means that the overhead of managing the futures comes to dominate the execution time. The solution to this problem is to increase the granularity of work, but not so much that there is not sufficient parallelism to keep the processors busy. In this paper, we introduce an automatic mechanism for chunking together fine-grain parallel computations into their sequential equivalents. Our approach, which is called hybrid chunking, uses a combination of compile-time analysis and runtime instrumentation and decision making to identify when a recursive parallel computation should be run as a recursive sequential computation.

We present a cost model that we use in the static analysis to obtain the cost of expressions and size information for data structures. We also describe how this information is used at runtime to switch the execution of a program from parallel to sequential when we determine it will run faster sequentially. We show that we can obtain good speedups on recursive divide-and-conquer programs, with our hybrid chunking approach.
Contents

Abstract 2

1 Introduction to Hybrid Chunking 1

2 Language L 4

3 Static Analysis 5
   3.1 Cost Model .................................................. 5
   3.2 Type system extension ..................................... 6
   3.3 Cost Rules ..................................................... 6
   3.4 Function Analysis ........................................... 9
      3.4.1 Higher Order Functions __________________________ 12
      3.4.2 Constraints ............................................... 13
      3.4.3 Recursive Functions .................................... 14
   3.5 Domain of the size and cost variables ...................... 15
      3.5.1 Sequential vs. Parallel Version ...................... 16
   3.6 Dynamic scheduling ......................................... 16

4 Implementation 17
   4.1 Assumptions .................................................. 18
   4.2 Static Component ............................................. 18
   4.3 Dynamic Component ......................................... 20
   4.4 Solving Recurrence Relations ................................. 21
   4.5 Limitations .................................................... 21

5 Example 23
   5.1 TreeAdd ....................................................... 23
      5.1.1 Static Analysis ........................................... 24
      5.1.2 Dynamic Decision ........................................ 25
   5.2 Listsplit ....................................................... 25
      5.2.1 Static Analysis ........................................... 26
   5.3 Compiler ....................................................... 28
      5.3.1 Static Analysis ........................................... 29
   5.4 Benchmarks .................................................... 31
      5.4.1 Threshold $T_c$ .......................................... 31
      5.4.2 TreeAdd .................................................... 35
## Contents

5.4.3 Optimized TreeAdd ................................................. 35  
5.4.4 Listsplit .............................................................. 41  
5.4.5 Compiler .............................................................. 42  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Related Work</td>
<td>44</td>
</tr>
<tr>
<td>Conclusion</td>
<td>46</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Granularity Problem .................................................... 2
1.2 Parallel Overhead ...................................................... 3
2.1 language L ............................................................... 4
3.1 Type System Extension and Cost Expressions ..................... 6
3.2 Subtyping Rules ......................................................... 7
3.3 Cost Rules ............................................................... 8
3.4 Function and Apply Cost Rules .................................... 10
3.5 Size of types ........................................................... 11
3.6 Free Variable Function ............................................... 12
5.1 Threshold TreeAdd .................................................... 32
5.2 Threshold Compiler .................................................... 33
5.3 Benchmark TreeAdd ................................................... 35
5.4 Benchmark TreeAdd Optimized .................................... 37
5.5 Benchmark TreeAdd Unbalanced Ratio 5:3:2 ..................... 39
5.6 Benchmark TreeAdd Unbalanced Ratio 8:1:1 ..................... 40
5.7 Benchmark Listsplit ................................................... 41
5.8 Benchmark Compiler .................................................. 43
1 Introduction to Hybrid Chunking

In recent years the computer industry shifted towards building multicore processors instead of trying to create faster single core machines. In order to take advantage of these additional cores, the industry has also reconsidered the processor architecture, the structure of the programming languages, and the ways that the programmer may take advantage of the additional cores. The decision of when to use parallel processing is a challenging question. It is difficult to evaluate if the overhead required to create the parallel task is greater than the benefit of computing an expression on multiple cores.

As seen in Figure 1.2, the overhead of creating the parallel version of the computation has to be taken into account. The parallel computation has to be big enough so that the overhead $O = O_{fork} + O_{join}$ does not slow down the computation. In the left picture the sequential computation is slower than the parallel computation of $e_1$ and $e_2$ including the overhead $O$. The right side of the figure illustrates a scenario in which sequential computation is faster than the parallel one [11].

This paper concentrates on that very question, namely when it is optimal to execute an expression sequentially rather than in parallel. The goal is to determine the point at which to switch from parallel to sequential computation in order to optimize the performance and minimize the overhead of running a task in parallel.

The Figure 1.1 shows the granularity problem. Granularity is the amount of computation in relation to communication. Communication in parallel execution of code is the time needed for the fork-join and possible synchronization of parts of the code. The chunks are individual tasks in terms of code size and execution time. The chart shows the runtime of a program with different chunk sizes. On the left hand side we can see that with a very small chunks the overhead of communication is big and the runtime is large too. On the right hand side we can see the chunk size being the entire program there is no parallelism and execution time is high but overhead is low. If we have multiple processors and we do not chunk the problem, we will leave other processors idle and waste computational resources. In order to find a good balance between communication and computational time, we want the chunk size to be in the red box.

To achieve the goal of a good chunk size we use an approach called hybrid chunking. Hybrid chunking refers to the strategy of static analysis of a program and the dynamic decision at runtime. The static analysis will analyze a given program and assigns cost to all expressions, if possible and size information to all datatypes. Size information represent how expensive operations on a datatsructure are. We create cost functions where we encounter recursive functions that depend on the size of the input arguments. The assigned cost represent an estimation of
the cost of execution of a part of code and the size information are used for functions if their runtime depends on the size of the input arguments. At runtime we dynamically make several decisions related to the sizing of the chunks of the work. We take the cost and size information assigned by the static analysis and evaluate how expensive the computation of a given part of code is. Based on that estimation we make a decision about the size of the chunks for that part of the code.

One class of problems that would benefit from dynamic chunking are divide and conquer algorithms. We want to dynamically identify potential recursive divide and conquer functions that can benefit from hybrid chunking during compile time. Also we want to make a decision about how much parallelism to introduce during the runtime of the program. We want to switch the computation from parallel to sequential at the moment where the costs for parallel computation are higher than the sequential costs. The appropriate number of chunks can also depend on the number of processors on a given machine and the workload of each of them at runtime.

We use a parallel tuple (PTuple) which has the form "(| expr1, expr2, ⋅⋅⋅ |)" to identify candidates for chunking. A PTuple is the parallel version of the SML tuple expression and allows the computation on the right hand side being executed in parallel.

```plaintext
fun addtwo(x) = x + 2

fun myadd (a,b) = let
val (x,y) = (| addtwo(a), addtwo(b) |)
    in
```

Figure 1.1: Granularity Problem
In this example the function myadd contains the PTuple \(| \text{addtwo}(a), \text{addtwo}(b) |\) which tells the compiler that the calls \(\text{addtwo}(a)\) and \(\text{addtwo}(b)\) are good candidates to be executed in parallel. This notation provided by the programmer does not require that the actual compiled code is run in parallel later.

This work is not introducing any new parallelism in a given program but instead optimizes existing parallelism to reduce overhead of the parallel computation.
2 Language L

In order to define the cost model, we first define a simple functional language L. The syntax of language L is given in Figure 2.1.

\[
\begin{align*}
x \in \text{var} & ::= x, y, z, ..., +, -, *, : \\
n \in \text{num} & ::= 0, 1, 2, ... \\
b \in \text{bool} & ::= \text{true}, \text{false} \\
\tau \in \text{tyexpr} & ::= \text{Unit} | \text{Int} | \text{Bool} \\
& | \tau_1 \rightarrow \tau_2 | \tau_1 \times \tau_2 | \tau_1 + \tau_2 \\
& | \text{List} \tau \\
e \in \text{expr} & ::= x | n | b | () \\
& | \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
& | \text{let } x = e_1 \text{ in } e_2 \\
& | \text{fun } f x = e_1 \text{ in } e_2 \\
& | e_1 \oplus e_2 \\
& | (e_1, e_2) \\
& | ([e_1, e_2]) \\
& | \text{case}(e_1, x, e_2, y, e_3) \\
& | e_1 :: e_2 \\
& | \text{Inl } e | \text{Inr } e \\
& | \text{Fst } e | \text{Snd } e
\end{align*}
\]

Figure 2.1: language L

The language L is a very simple higher-order, monomorphic and call-by-value language. The fun expression allows us to write functions and especially recursive functions. The function f can be called in \( e_1 \) and \( e_2 \). The expression \( e_1 \oplus e_2 \) is an application. It applies a function \( e_1 \) to an argument \( e_2 \). We have a tuple expression of the form \( (e_1, e_2) \) and the parallelism is introduced in form of a parallel tuple \( ([e_1, e_2]) \). All expressions in a ptuple can be executed in parallel. We have sum types in form of the case expression \( \text{case}(e_1, x, e_2, y, e_3) \), \( \text{Inl } e \) and \( \text{Inr } e \). The case expression evaluates \( e_1 \) and depending on the result is \( x \) or \( y \) it executes \( e_2 \) or \( e_3 \). The \( \text{Inl } e \) and \( \text{Inr } e \) take a sum type and return the left or the right side of the sum type and the \( e_1 :: e_2 \) expression is the list constructor. We predefine the operators +, -, *, and : as variables.
3 Static Analysis

Our model uses both static and a dynamic analysis to determine the chunking decision. The static portion analyzes a program at compile time and the dynamic process optimizes at runtime. At compile time we collect cost and size information about expressions and use them to generate cost functions. During the dynamic analysis we use these informations to make a decision about the chunking. In this chapter we introduce a cost model for our language L that contains two parts, cost evaluations and size determinations. Each expression in our language has a cost attached to it, while each datatype, like integers, booleans, list or user defined data types, gets its size attached. Size represent how expensive a computation on the datastructure is. The cost of user defined functions may depend on the input argument.

3.1 Cost Model

In order to perform hybrid chunking, we have to define a model that gives us information about sizes of data and corresponding cost that we can assign to the expressions in the static analysis. Since we are especially interested in performing chunking on recursive functions and corresponding recursive data types, we need to obtain size information about these data structures and types in order to compute the cost of computations. For example, consider the cost of the simple operation:

\[ \text{fun } \text{add } (x) = x + 5 \]

If we want to be as precise as possible, we have to account for the cost of the operation on the given data (\textit{i.e.}, the operator \(+\)), the cost of looking up \(x\), and the cost of the integer \(5\). We decided to ignore the cost for looking up \(x\) and \(5\), and just charge for \(\beta\)-reductions as proposed by Loidl and Hammond [13]. Since it is impossible to get an exact cost, we would like to find a way to compute an upper bound on the cost of a computation or function.

First we extend the type system with size and cost values and variables. Then we use the defined size and cost values and variables to gather information about expressions and types during the static analysis.
3 Static Analysis

3.2 Type system extension

For our defined language L, we modify the original type system to account for size information (see Figure 3.1) and our cost expressions are defined in Figure 3.1. Sizes are integers that give us information about how big a datastructure is. We need sizes to measure the cost of operations on datastructures for example to sum up all elements in a list, we need to know how many elements are in the list. The cost are also integers and represent how expensive a computation of an expression is. Cost are used to account for latent costs.

\[
\begin{align*}
\tau &::= \text{Unit}^1 \mid \text{Int}^\text{size} \mid \text{Bool}^1 \\
&\mid \tau_1 \xrightarrow{c} \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \\
&\mid \text{List}\text{\(\tau\)}^\text{size} \\
\end{align*}
\]

\[
\begin{align*}
c &::= n \mid c_1 + c_2 \mid c_1 - c_2 \mid c_1 \times c_2 \\
&\mid \max(c_1, c_2) \mid c_f(c_1 \ldots c_n) \\
\end{align*}
\]

\[\text{size} ::= 0, 1, \ldots, \infty\]

In the cost expressions, n is a positive integer. The last cost expression \(c_f(c_1 \ldots c_n)\) represents a cost function. We need a cost function if the cost of a function \(f\) depends on the cost of the input arguments \(c_1 \ldots c_n\). This way the cost of the function can be computed as soon as the input arguments are known. We define size + 1 to be the successor of size and the operation \(\infty + 1 = \infty\).

3.3 Cost Rules

In order to capture size and cost information in the type system we want to introduce cost rules. A cost rule has the form \(\Gamma \vdash e : \tau \parallel c\). Where \(\Gamma\) is a type assignment or type environment, which is a finite map from variables to types. The type environment \(\Gamma\) is used to look up the types of the free variables, \(e\) is an expression, \(\tau\) is the size type of \(e\), and \(c\) is the cost of evaluating \(e\). We write \(\Gamma[x : \tau]\) to mean the environment \(\Gamma\) extended with the binding of \(\tau\) to \(x\). We introduce cost rules in this section except for functions and the apply expression, which we introduce in Section 3.4.

In order to define the cost rules, we have to define subtyping rules first. The size and cost are an upper bound and therefore it is possible to overestimate given size or cost even more. Figure 3.2 shows the rules for relaxing the upper cost and size bounds for an expression in our language L. Figure 3.2 shows the rules for relaxing the upper cost and size bounds for an expression in our language language L. This definition leads to the cost rules for the expressions in our language in the type system defined in Figure 3.3.

The assignment of the cost to the different expressions will give us a good prediction of how
3 Static Analysis

\[
\begin{align*}
\frac{\tau_1 < \tau_1 \quad \tau_2 < \tau_2 \quad c < c'}{
\tau_1 \rightarrow \tau_2 < \tau_1' \rightarrow \tau_2'} \\
\frac{\tau_1 < \tau_1' \quad \tau_2 < \tau_2'}{
\tau_1 \times \tau_2 < \tau_1' \times \tau_2'} \\
\frac{\tau_1 < \tau_1' \quad \tau_2 < \tau_2'}{
\tau_1 + \tau_2 < \tau_1' + \tau_2'} \\
\frac{\text{size} < \text{size}'}{
\text{Int}^{\text{size}} < \text{Int}^{\text{size}'} \\
\frac{\text{size} < \text{size} \quad \tau_1 < \tau_1'}{
\text{List}^{\text{size}} \tau_1 < \text{List}^{\text{size}'} \tau_1'} \\
\tau < \tau \\
\text{size} < \text{size} \\
\text{size} < \infty \\
\frac{c_1 < c_1' \quad c_2 < c_2'}{
\frac{c_1 + c_2 < c_1' + c_2'}{c_1 - c_2 < c_1' - c_2'} \\
\frac{c_1 \times c_2 < c_1' \times c_2'}{c_1 < c_1' \quad c_2 < c_2'} \\
\frac{\text{max}(c_1, c_2) < \text{max}(c_1', c_2')}{c < \infty} \\
\frac{\text{c} < \text{c}'}{c < \infty}
\end{align*}
\]

Figure 3.2: Subtyping Rules
3 Static Analysis

(Int) \[ \Gamma \vdash n : \text{Int} \parallel 0 \]

(Bool) \[ \Gamma \vdash b : \text{Bool} \parallel 0 \]

(Cons) \[ \begin{aligned} &\Gamma \vdash e_1 : \tau \parallel c_1 \\
&\Gamma \vdash e_2 : \text{List}^{\text{size}_\tau} \parallel c_2 \\
&\Gamma \vdash e_1 :: e_2 : \text{List}^{\text{size}_\tau + 1} \parallel 1 + c_1 + c_2 \end{aligned} \]

(Nil) \[ \Gamma \vdash \text{nil} : \text{Unit} \parallel 0 \]

(Var) \[ \Gamma \vdash x : \tau \parallel 0 \]

(Tuple) \[ \begin{aligned} &\Gamma \vdash e_1 : \tau_1 \parallel c_1 \\
&\Gamma \vdash e_2 : \tau_2 \parallel c_2 \\
&\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \parallel c_1 + c_2 + 1 \end{aligned} \]

(Parallel Tuple) \[ \begin{aligned} &\Gamma \vdash e_1 : \tau_1 \parallel c_1 \\
&\Gamma \vdash e_2 : \tau_2 \parallel c_2 \\
&\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \parallel 1 + \text{max}(c_1, c_2) \end{aligned} \]

(if) \[ \begin{aligned} &\Gamma \vdash e_1 : \text{Bool} \parallel c_1 \\
&\Gamma \vdash e_2 : \tau \parallel c_2 \\
&\Gamma \vdash e_3 : \tau \parallel c_3 \\
&\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \parallel 1 + c_1 + \text{max}(c_2, c_3) \end{aligned} \]

(let) \[ \begin{aligned} &\Gamma \vdash e_1 : \tau_1 \parallel c_1 \\
&\Gamma \vdash \text{x} : \tau_1 \parallel 0 \\
&\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \parallel 1 + c_1 + c_2 \end{aligned} \]

(Case) \[ \begin{aligned} &\Gamma \vdash e_1 : \tau_1 + \tau_2 \parallel c_1 \\
&\Gamma \vdash \text{x} : \tau_1 \parallel 0 \\
&\Gamma \vdash e_2 : \tau_3 \parallel c_2 \\
&\Gamma \vdash \text{y} : \tau_2 \parallel 0 \\
&\Gamma \vdash e_3 : \tau_3 \parallel c_3 \\
&\Gamma \vdash \text{case}(e_1, x, e_2, y, e_3) : \tau_3 \parallel 1 + c_1 + \text{max}(c_2, c_3) \end{aligned} \]

(Inl) \[ \begin{aligned} &\Gamma \vdash e : \tau_1 \parallel c \\
&\Gamma \vdash \text{Inl } e : \tau_1 + \tau_2 \parallel 1 + c \end{aligned} \]

(Inr) \[ \begin{aligned} &\Gamma \vdash e : \tau_2 \parallel c \\
&\Gamma \vdash \text{Inr } e : \tau_1 + \tau_2 \parallel 1 + c \end{aligned} \]

(Fst) \[ \begin{aligned} &\Gamma \vdash e : \tau_1 \star \tau_2 \parallel c \\
&\Gamma \vdash \text{Fst } e : \tau_1 \star \tau_2 \rightarrow \tau_1 \parallel 1 + c \end{aligned} \]

(Snd) \[ \begin{aligned} &\Gamma \vdash e : \tau_1 \star \tau_2 \parallel c \\
&\Gamma \vdash \text{Snd } e : \tau_1 \star \tau_2 \rightarrow \tau_2 \parallel 1 + c \end{aligned} \]

(Sub) \[ \begin{aligned} &\Gamma \vdash e : \tau \parallel c \\
&\tau \prec \tau' \\
&\Gamma \vdash e : \tau' \parallel c \end{aligned} \]

Figure 3.3: Cost Rules
expensive the computation of the different expressions might be.

3.4 Function Analysis

We are especially interested in recursive functions that compute over recursive data structures. These functions will benefit most from our hybrid chunking approach, but they are also the most complex to analyze. We start with an explanation of how to obtain the cost of a simple function and move on to the more complex recursive case.

In order to obtain the cost of a function, we have to analyze its complexity and the dependency of the cost on the input arguments. The first step has to be a sensitivity analysis regarding the input data. If a function does not depend on its input arguments it is fairly easy to obtain a cost function, or even a fixed cost term. Let us start with a function where we can obtain a cost that does not depend on the input argument.

```plaintext
fun test(x) = if ( x = 0 ) then x + 1 else 5
```

This function is a simple example and it is easy to see that its cost does not depend on x. In this function test we can look up the type of the variable x in the environment which will tell us that x is of type int. As we recall from Section 3.3, we know that an integer has cost 0 and the if statement will give us the cost $1 + \max(1, 0) = 2$, which will be the cost of the function test.

The following implementation of the fib function is a simple case of a function that is data sensitive.

```plaintext
fun fib 0 = 0 |
  fib 1 = 1 |
  fib n = fib (n-1) + fib (n-2)
```

The running time is an exponential and determined by the input argument. The cost of the fib function depends on a non-recursive data type and therefore it is easy to compute the cost. The cost function our static analysis creates for the function fib will take the size of an integer as an input argument. As defined in Figure 3.3 the size of integer is the integer itself. The cost function fib cost looks like the following.

```plaintext
fun costfib inputsize = if (inputsize < 2) then 0 else 1 + costfib(inputsize - 1) + costfib(inputsize - 2)
```

The treeAdd function is an example of a recursive function with a recursive data type as input argument. The goal of our analysis is to find a cost function for such a recursive functions.

```plaintext
datatype tree = Lf of int |
  Nd of tree * tree

fun treeAdd (Lf n ) = n
```
3 Static Analysis

\[
\text{let } (x, y) = (\text{treeAdd (tL), treeAdd (tR)}) \text{ in } x + y \text{ end}
\]

From these examples we can already see how the cost rules for functions and the corresponding application expression look like, they are defined in Figure 3.4.

(Assume that Figure 3.5 defines the size function \(| | \).

\[
\begin{align*}
\frac{\Gamma[f : \tau_1 \xrightarrow{c_f} \tau_2, x : \tau_1] \vdash e_1 : \tau_2 \parallel c_f}{\Gamma \vdash \text{fun } f x = e_1 \text{ in } e_2 : \tau \parallel 1 + c_f} & \quad \text{if } c_f \text{ is a fixed cost term} \\
\frac{\Gamma \vdash e_1 : \tau_1 \parallel c_1 \quad \Gamma \vdash e_2 : \tau_1 \parallel c_2}{\Gamma \vdash e_1 \oplus e_2 : \tau_2 \parallel 1 + c_f + c_1 + c_2} & \quad \text{if } c_f \text{ is a fixed cost term} \\
\frac{\Gamma[f : \tau_1 \xrightarrow{c_f} \tau_2, x : \tau_1] \vdash e_1 : \tau_2 \parallel c_f(|\tau_1|)}{\Gamma \vdash \text{fun } f x = e_1 \text{ in } e_2 : \tau \parallel 1 + c_f} & \quad \text{if } c_f \text{ is a cost function} \\
\frac{\Gamma \vdash e_1 : \tau_1 \parallel c_1 \quad \Gamma \vdash e_2 : \tau_1 \parallel c_2}{\Gamma \vdash e_1 \oplus e_2 : \tau_2 \parallel 1 + c_1 + c_2 + c_f(|\tau_1|)} & \quad \text{if } c_f \text{ is a cost function}
\end{align*}
\]

Figure 3.4: Function and Apply Cost Rules

A function \(f\) gets either a fixed cost or a cost function attached to it. We determine this by analyzing \(e_1\) and \(e_2\) of the function \(f\). The last two rules in Figure 3.4 cover the case of a cost function. The cost function has two parts. The first part is the fix cost of the expressions \(e_1\) and \(e_2\) and the second part is the recursive call to the cost function itself. The recursive part corresponds to the recursive call in the function \(f\). We can obtain the cost of a function call in the apply expression by applying the size of the input argument of the function \(f\) to the cost function \(c\). The cost function \(c\) was created because the function \(f\) depends on the size of its input arguments. The correct cost for executing the function will be the size of \(\tau_1\), which we get from applying the size function \(||\) defined in Figure 3.5 on the type \(\tau_1\) of \(e_2\) applied to the cost function \(c_f\) and plus one for the apply operation itself and \(c_1 + c_2\) for evaluating \(e_1\) and \(e_2\).

From looking at these examples we can obtain rules that classify the data sensitivity of the different expressions in our language L.

\[
\begin{align*}
\text{fun } & fx = e_1 \text{ in } e_2 \\
\text{f} = \left\{ \begin{array}{ll}
\text{data sensitive} & \text{if } x \in \text{FV}(e_1) \\
\text{not data sensitive} & \text{otherwise}
\end{array} \right.
\end{align*}
\]
3 Static Analysis

\[ |Int| = size \]
\[ |Bool| = 1 \]
\[ |Unit| = 1 \]
\[ |List| = size \]
\[ |\tau_1 \times \tau_2| = |\tau_1| + |\tau_2| \]
\[ |\tau_1 + \tau_2| = |\tau_1| + |\tau_2| \]
\[ |\tau_1 \to \tau_2| = |\tau_2| \]

Figure 3.5: Size of types

FV is a function that computes the set of free variables in an expression and is defined in Figure 3.6. It takes an expression of the language Land returns a set of free variables in that expressions. A free variable \( v \) is a variable within an expression \( e_1 \) that is not bound to any expression, including \( e_2 \), in the scope of the expression \( e_1 \).

We choose a conservative definition of data sensitivity in order minimize mismatches. There can be cases where a free variable appears in an expression but it is not used in a way that the expression depends on it. For example,

\[
\text{fun } \text{datasens} \ (x) = \text{let} \ \\
\quad \text{val } y = x \ \\
\quad \text{in} \ \\
\quad 2 \ \\
\quad \text{end}
\]

the function \text{datasens} will be recognized as data sensitive by our definition due to the fact that \( x \) is a free variable in the body of \text{datasens} but it is not used. So in fact \text{datasens} is data insensitive.

We can match functions to the data sensitivity rule in order to make a decision and develop an inference system. It might be impossible to obtain the cost for functions that depend on the size of the input arguments. If we cannot find a cost function for a data sensitive function with pattern matching, we should define it as unknown. We do not make chunking decisions for unknown functions.

For example the following version of treeAdd with a call to a fib function is difficult to analyze.

\[
\text{datatype } \text{tree} = \text{L}f \text{ of } \text{int} \ \\
\quad | \text{N}d \text{ of } \text{tree} \times \text{tree}
\]
FV(x) = \{x\}
FV(n) = \{\}
FV(b) = \{\}
FV(let x = e1 in e2) = FV(e1) \cup FV(e2) \setminus \{x\}
FV(case(e1, x, e2, y, e3)) = FV(e1) \cup FV(e2) \setminus \{x\} \cup FV(e3) \setminus \{y\}
FV(if e1 then e2 else e3) = FV(e1) \cup FV(e2) \cup FV(e3)
FV(e1 @ e2) = FV(e1) \cup FV(e2)
FV(fun fx = e1 in e2) = FV(e2)
FV((e1, e2)) = FV(e1) \cup FV(e2)
FV(Inl e) = FV(e)
FV(Inr e) = FV(e)
FV(Snd e) = FV(e2)

Figure 3.6: Free Variable Function

fun fib 0 = 0
| fib 1 = 1
| fib n = fib(n-1) + fib(n-2)

fun treeAdd (Lf n ) = fib(n)
| treeAdd (Nd (tL, tR)) = let
  val (x,y) = (| treeAdd (tL), treeAdd (tR) |)
  in
  x + y
end

Since we do not know the value of n at compile time we cannot solve the dependency between
the Leaf as the input argument to \texttt{treeAdd} and the number on the leaf to the \texttt{fib} function.
We will mark the function as unknown. The function \texttt{treeAdd} will get a cost of $T_u$ during the
static analysis as defined in Section 3.5.

3.4.1 Higher Order Functions

It might be difficult to obtain cost for higher order functions. For example:

let
  fun f (x) = x + 2
  fun g (j) = j(2)
in
  val t = g (f)
end

In the function g, it is not obvious what the cost or type for g is since the input j is another
function. We have to analyze the function j and get a relation between the possible input ar-
3 Static Analysis

Arguments and output arguments in order to know more about the cost of evaluation. We can use techniques like control flow analysis to analyze the calling sites of \( g \) to find out the correct cost. In order to capture the cost of the higher-order function \( g \) we attach latent cost to it [15]. To stay with the example, the first thing to analyze are the types of \( f \) and \( g \).

\[
f: \text{Int}^\infty \xrightarrow{c_f} \text{Int}^\infty \\
g: \tau_1 \xrightarrow{c_g} \tau_2.
\]

If we analyze the cost for the functions we can see that \( f \) has one primitive operation and two integers. Integers have cost 0 and the one primitive operation has cost 1 which gives us \( c_f = 1 \) and therefore \( f: \text{Int}^\infty \xrightarrow{1} \text{Int}^\infty \).

The cost for \( g \) depend on the input function \( j \). That’s why we have to create a cost function \( c_g(j) = c_j + 1 \), where \( c_j \) are the cost of the function \( g \) with the cost input argument \( j \) and the cost of \( g \) depends on the cost of \( j() \) and the cost for a function application, which is 1.

With control flow analysis we can find out that the only calling function to \( g \) is actually \( f \) and therefore the cost and type representation changes to

\[
f: \text{Int}^\infty \xrightarrow{1} \text{Int}^\infty \text{ and } \\
g: (\text{Int}^\infty \xrightarrow{1} \text{Int}^\infty)^2 \xrightarrow{2} \text{Int}^\infty
\]

3.4.2 Constraints

In order to compute the cost of an expression that might be run in parallel we want to collect constraints for the function expression \( \text{fun } fx = e_1 \text{ in } e_2 \) [4]. The constraints that catch the type size and cost dependencies between the input and output variables of a function need to be captured and attached to the function. For example:

\[
\text{fun sum nil = 0} \\
| \text{sum (x::xs) = x + sum (xs)}
\]

The function \( \text{sum} \) sums up all the elements in the list and has the type:

\[
\text{val sum: } (\text{List}^{\infty} \tau_1) \xrightarrow{c_\text{sum}(L)} \tau_2
\]

The \( L \) represents the length of the input list, \( \tau_1 \) and \( \tau_2 \) are the size type variables that represent the size type of the elements of the list and output of the function, which will be integers in our language since we do not support any other numeric types in our language \( L \). This makes the type of \( \text{sum} \)

\[
\text{val sum: } (\text{List}^{\infty} \text{Int}^{\infty}) \xrightarrow{c_\text{sum}(L)} \text{Int}^{\infty}
\]

with \( n \) being the maximum of all integers in the List and \( m \) the size of the output integer. The \( c_\text{sum} \) represents the cost of the function \( \text{sum} \) that depend on the length \( l \) of the input list, therefore the cost expression \( c_\text{sum}(L) \) will be of the form \( c_\text{sum}(L) = L \cdot c \), with \( c \) being a constant factor.

We will get to the closed form expression of recursive functions in the Chapter 3.4.3. We can add the following constraint to the sum function:

\[
n \leq m
\]
3 Static Analysis

In order to find a solution to the cost problem we have to solve the constraint system. We might need additional techniques like control flow analysis to obtain cost constraint for higher order functions and optimization of uncalled functions.

3.4.3 Recursive Functions

We can solve cost expressions for all the expression forms in our language except for recursive functions. For recursive functions we do not get a closed form expression for the cost, which means we will have a reference to a cost function in the cost expression. Therefore we need a way to solve a recurrence relation in a constraint set. For example the cost function of the `treeAdd` function from 3.4 will look like this.

```
fun treeAddcost (sizeofinputtree) =
  if (sizeofinputtree <= 1)
    then 0
  else 2 * treeAddcost(sizeofinputtree / 2)
```

Where `sizeofinputtree` is the size of the tree that `treeAdd` gets called on and `c` are the fix costs of the `treeAdd` function. We have to assume that the tree is balanced, which means that the left and right subtree of a Node have approximately the same size. With this assumption we can create the recursive call `2 * treeAddcost(sizeofinputtree/2)` in the body of the cost function. Solving this recurrence relation will make it easier to compute the cost of a function call dynamically at runtime.

We want to use a “database” to solve recurrence relations. The following table shows some recurrences and their closed form expressions.

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Closed Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 = a$</td>
<td>$f_1 = a$</td>
</tr>
<tr>
<td>$f_n = b + f(n - 1)$</td>
<td>$f_n = a + b \cdot n$</td>
</tr>
<tr>
<td>$f_1 = a$</td>
<td>$f_1 = a$</td>
</tr>
<tr>
<td>$f_n = b + c \cdot f(n - 1)$</td>
<td>$f_n = a + c \cdot n + b \cdot \frac{c^{n-1}}{c-1}$</td>
</tr>
<tr>
<td>$f_1 = a$</td>
<td>$f_1 = a$</td>
</tr>
<tr>
<td>$f_n = b + c \cdot n + f(n - 1)$</td>
<td>$f_n = a + b \cdot n + c \cdot n \cdot \frac{c^{n} + 1}{c-1}$</td>
</tr>
<tr>
<td>$f_n = f(n - 1) + f(n - 2)$</td>
<td>$f_n = \lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \rfloor$</td>
</tr>
<tr>
<td>and $\phi = \frac{1 + \sqrt{5}}{2}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Closed Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 = a$</td>
<td>$f_1 = a$</td>
</tr>
<tr>
<td>$f_n = k \cdot f(n/b) + c$</td>
<td>$f_n = \frac{-c + a \cdot k \cdot log(n) + c \cdot k^1 \cdot log(n) + a \cdot k^1 \cdot log(n)}{1 + k}$ if $k \neq 1$</td>
</tr>
<tr>
<td>otherwise $f_n = \frac{a \cdot log(b) + c \cdot log(n)}{log(b)} = a + c \cdot log_b(n)$</td>
<td></td>
</tr>
</tbody>
</table>
3 Static Analysis

Especially the last case

\[ f \ 1 = a \\
| f \ n = k \ast f \ ( \ n / b ) + c \]

is relevant to our attempt to find closed cost functions to recursive functions. It will allow us to compute the function costs with a non recursive cost function.

3.5 Domain of the size and cost variables

The domain for cost variables are the natural numbers up to a threshold \( T_c \) where we know that parallel execution is the correct strategy. It is not necessary to know the cost beyond that point since we use the cost to make a decision between parallel and sequential execution. This strategy simplifies the implementation and allows us to use integers for cost and size bookkeeping. We will empirically define the threshold \( T_c \) with a set of experiments. Also \( \infty \) represents unbounded cost if the cost are higher than the threshold \( T_c \), which is not the same as the unknown cost. Finally we need a value for unknown cost \( T_u \).

\[ D_c = \{ 0, 1, \ldots, T_c, \infty, T_u \} \]

The size of the different expressions can be defined as following: Booleans have size 1, numbers have their actual value as their size and variables get a size variable attached to them. Numbers have the type Int or Int\( ^{v} \) with \( v \) being the actual value or upper bound on the number or numbers in a list. Variables have to get a var size since their size might be unknown. More interesting cases are recursive datatypes like lists. The length of the list is a natural choice for its size representation, \( List^{z} \). For example a list \([1, 2, 3, 4, 5]\) has size type \( List^{6} Int^{5} \). The list has size 6, 5 for its elements and 1 for the NIL at the end of it. The 5 on the Int is the highest number in the list. Other recursive data types can be charged the number of data types in the following recursion steps as cost. For example the tree datatype will have the number of subnodes as size and 1 for a leaf.

```
datatype tree = Lf of int
| Nd of tree * tree
```

The following tree

```
  .
 /|
/ |
1 2 3 4
```

gets size information
The Domain for size types is defined as the natural numbers:

\[ D_\tau = \{0, 1, \ldots, T_s\} \]

With \( T_s \) as an upper bound on the size. Since we are especially interested in the cost of recursive functions that depend on the size of their input arguments, we assume that we charge at least \( c = 1 \) for each size value. Therefore there exists a relationship between \( T_s \) and \( T_c \) and we can use the same threshold for the upper bound for size and cost \( T_s = T_c \).

### 3.5.1 Sequential vs. Parallel Version

The decision to run a function in parallel or sequentially depends on the cost of the two different versions and the overhead that we have to account for in the parallel case. In our previous example treeAdd

```haskell
fun treeAdd (Lf n) = n
| treeAdd (Nd (tL, tR)) = let
  val (x,y) = (| treeAdd (tL) , treeAdd (tR) |)
  in
  x + y
end
```

we have to consider the case when we run \texttt{treeAdd} in parallel and therefore the cost for executing \((| \texttt{treeAdd (tL)} , \texttt{treeAdd (tR)} |)\) is the same as executing one of the branches because they run in parallel. However we must account for the added branching costs. In the sequential version the cost will be the cost to evaluate \texttt{treeAdd} on the right and left subtree.

### 3.6 Dynamic scheduling

During the runtime of a program, we have to use additional information to decide how to split up work. Whether the chunks get executed in parallel will also depend on the number of idle processors or the number of processors with just a minimum amount of work. Since Manticore is built to create a lot of threads to take advantage of parallelism in the code and the available resources on a computer, it is possible that a certain number of processors are busy doing other computations when we find a piece of work that can be chunked dynamically and it will remain on one processor for execution.

We will discuss the mechanism for splitting up the work and distributing it on available resources in the implementation part of this paper in Section 4.
4 Implementation

The implementation of the hybrid chunking approach has two components: static and dynamic. The first component is the static analysis of a program. With the cost model presented in Section 3.2 we use a collecting semantic approach to collect cost information. At runtime we use a dynamic decision process to chunk the work in appropriate sizes that fit the available resources of the computer the program is running on.

For the implementation we use our research compiler Manticore. Manticore is a compiler for the parallel programming language Parallel ML (PML), which is based on Standard ML. PML supports parallelism using an implicitly-threaded model where the programmer annotates parts of the program that should be run in parallel. One of the explicit parallelism constructs is a PTuple. A PTuple expression is the parallel version of a tuple known from SML and indicates that all expression inside the tuple can be executed in parallel.

```ml
fun fib 0 = 0
  | fib 1 = 1
  | fib n = let
      val (a,b) = (| fib(n-1) , fib(n-2) |)
  in
      a+b
  end
```

This implementation of the `fibonacci` function contains the PTuple `(| fib(n-1) , fib(n-2) |)` which gives the hint that both values might be computed in parallel.

We implemented the cost model in the abstract syntax tree (AST) stage of the Manticore compiler, which is an explicitly-typed representation of the program. In this stage we still have explicit parallelism in form of a PTuple expression and all type information. In the AST stage we perform multiple optimization including flattening of nested-data-parallelism, introduction of futures for implicitly-threaded parallel constructs and compilation of pattern matching. All expressions in a PTuple compile to a future (the chunks we size with our hybrid chunking strategy) except the first. The first expression will be executed by the processor the entire function is scheduled on. Futures can be stolen by other processors due to the work stealing scheduler in Manticore. The work stealing allows for a good work distribution throughout the available resources on a machine and is an essential part of executing code in parallel. The `fibonacci` example will change into a representation where the PTuple is replaced with a future.

```ml
fun fib 0 = 0
  | fib 1 = 1
  | fib n = let
```
4 Implementation

\[
\text{val } (a, b) = \text{let}
\begin{align*}
\text{val } f1 &= \text{fib}(n-1) \\
\text{val } f2 &= \text{future} ( \text{fn } _ => \text{fib}(n-2) ) \\
\text{in}
(f1, \text{touch } f2) \\
\text{handle } e => (\text{cancel } f2; \text{raise } e)
\end{align*}
\text{in}
(a + b)
\text{end}
\]

4.1 Assumptions

We specifically target recursive functions and recursive data types for the hybrid chunking. Tests indicate that in order for this approach to benefit a user program, the recursive function must execute an extended time. For this reason we optimize if we find a PTuple expression inside a recursive function that contains calls to the function itself. Hybrid chunking is especially useful for functions that depend on the size of the input argument.

We assume that data structures, like a tree, are balanced, which means that all subtrees of a node have the same size. This assumption is reasonable because an unbalanced data structure would yield no benefit by being run in parallel. We will show that this assumption is correct in the chapter 5.4.3. This is due to the delay required for one thread to wait until another corresponding thread with a larger subtree was executed. Additionally the join fork overhead would introduce a penalty compared to the sequential execution.

4.2 Static Component

We perform multiple passes on the representation of the program. All passes are performed at the end of the AST stage, after optimizations, of the compiler.

In the first pass we collect all type constructors that are used in the program. Type constructors are used to create the data constructors in a program.

\[
\text{datatype } \text{tree } = \text{Leaf } \text{of } \text{int} \\
| \text{Node } \text{of } \text{tree } * \text{tree}
\]

The type constructor in this example is tree and the data constructors are Leaf and Node. There is a one to one relationship between the type and data constructor representation in Manticore. Each type constructor has a list of the data constructors of the type and the data constructor contain a pointer back to the type constructor. This allows us to scan for the occurrences in the code where the data constructors are used inside functions and PTuple expressions.
The next pass collects cost information about all expressions according to Figure 3.3. Wherever we find a function that has a call to itself, we create a cost function. We use variable annotations to mark a function so we know to create a cost function at the end of the function analysis.

```haskell
fun fib 0 = 0
    | fib 1 = 1
    | fib n = fib (n-1) + fib (n-2)
```

When we analyze the fibonacci function we mark it as recursive to create a cost function. We mark the function name, which is the variable fib, with a mapping $fib \rightarrow attribute$. We also save all the calls to functions outside the current function scope.

```haskell
fun treeAdd (Lf n) = n
    | treeAdd (Nd (tL, tR)) = let
      val (x,y) = (| treeAdd (tL), treeAdd (tR) |)
      val _ = fib(5)
      in
        x + y
      end
```

We save the information that we have a call to fib in the `treeAdd` function. We have to check if there exists a cost function for fib and whether the input argument is open or like the integer 5 a closed expression. If it is closed we can go ahead and create the cost function for treeadd.

In the next pass we search for PTuple expressions in the body of functions that use a type constructor and have a cost function attached. When we encounter such a function, we mark the type constructor as used so we can add size information to it. The PTuple expression is changed to an if statement that uses the size information of the arguments in the apply expression inside the PTuple to obtain the cost.

```haskell
val (x,y) = (| treeAdd (tL), treeAdd (tR) |)
```

changes to

```haskell
val (x,y) = if ( treeAddcost ( sizetree(tL) ) +
                 treeAddcost ( sizetree(tR) ) > Tc )
  then (| treeAdd (tL), treeAdd (tR) |)
  else ( treeAdd (tL), treeAdd (tR) )
```

If we find functions that use size information for the dynamic chunking we have to change the type and data constructor in the next pass. The type tree for example changes dynamically to account for size information.

```haskell
datatype tree = Leaf of int
    | Node of tree * tree
```

changes to

```haskell
datatype tree = Leaf of int
    | Node of int * tree * tree
```
4 Implementation

Where the first integer field of the data constructors contains the size information, we do not need to expand the base case since we assume it has size 1. In order to fill in size information at time of creation of the data structure, we also have to manipulate the function that creates the data structure and generate a function that returns the size information for the specific data type.

```
fun mkTree d = let
    fun mk d' = if d' >= d
        then T.LEAF (1)
        else T.NODE ( | mk (d'+1), mk (d'+1) |)
    in
        mk 0
    end
```

The function `mkTree` creates a tree that contains a one in the leaf and has depth d. To account for size information the function changes in the backend of the compiler to the following representation.

```
fun mkTree d = let
    fun mk d' = if d' >= d
        then T.LEAF (1)
        else let
            val (tree1, tree2) = ( | mk (d'+1), mk (d'+1) |)
            val size = sizetree(tree1) + sizetree(tree2) + 1
        in
            T.NODE(size, tree1, tree2)
        end
    in
        mk 0
    end
```

(* returns the size of the tree *)

```
fun sizetree (Lf (_) ) = 1
    | sizetree (Nd (size, _, _ ) = size)
```

The size function can easily be created with the information of the type constructor.

With all this information added to the user program we are able to make a dynamic decision at runtime.

4.3 Dynamic Component

We have to consider the cost of fork and join compared to running a program sequentially and find the right size for the chunks. After collecting the information in the static phase we know have enough information to decide how to size the chunks of work properly. Manticore follows the approach of creating as much parallelism and threads as possible to keep all processors busy at any given time. Manticore uses a work stealing approach to achieve a load balancing of work
4 Implementation

throughout all processors. We want to make sure that chunking the work in the correct size allows us to keep an even level of work for all processors.

The runtime takes care of resource management and mapping available processors to the different work chunks. This is part of the Manticore runtime and scheduler [8], [9] and [7].

4.4 Solving Recurrence Relations

During the implementation we determined that using the closed form expression of recurrence relations is actually slower than leaving the recursive function in its original form. Especially the following closed form expression is used in our code.

\[
\begin{align*}
    f_1 &= a \\
    f_n &= k \cdot f\left(\frac{n}{b}\right) + c
\end{align*}
\]

\[
\Rightarrow f_n = \frac{-c - a \cdot k^{\log_b(n)} + c \cdot k^{1+\log_b(n)} + a \cdot k^{1+\log_b(n)}}{1+k} \quad \text{if } k \neq 1
\]

otherwise \( f_n = \frac{a \cdot \log(b) + c \cdot \log(n)}{\log(b)} = a + c \cdot \log_b(n) \)

We identified two important factors why the closed form expression is slower than the recursive function call. First of all, due to the implementation of the basis library in Manticore that contains the logarithm and exponential function we lose a lot of performance. The basis library calls a C function in order to compute the values which is an expensive operation. The second reason is that the threshold for the cutting of value for sizes and costs is fairly small, in the examples in chapter 5 we use 10000. That means, if we take a threshold of 10000, we have at most 14 recursive calls, assuming that \( n_{max} = 10000 \) and \( b_{min} = 2 \) which leads to the formula \( 2^k = 10000 \) assuming we stop the recursion when the size is less than one, which means \( k \leq 14 \).

4.5 Limitations

We have multiple limitations due to the AST stage and different assumptions of the cost model.

So far our implementation only recognizes user defined data types that are used in a recursive function that contains a PTuple. We discard higher order functions due to the difficulties in performing a proper analysis on the call sites of a function in the AST stage. In order to take advantage of the parallel infrastructure of Manticore we have to implement the static analysis in the AST phase because we discard type information in the transformation between AST and the following BOM stage.

We can only create cost functions in the static analysis for functions that depend on a single input variable because it might not be possible to identify the dependent variable for the cost function. To support multiple input variables, we have to perform data flow analysis. This is
very difficult in the AST stage for the same reason we do not perform control flow analysis: the
AST phase makes it very difficult to implement these analysis. We do not have all the unification
transformation in the AST stage nor the infrastructure to attach information to expressions. Also
not all expressions are bound to variables. This step is done in the BOM stage, which does not
contain type information anymore.

We assume that the threshold $T_c$ for the cut off of the parallel computation is fixed for all pro-
grams. We do not use a profiler to recompute it for different systems or examples. This might
lead to slightly slower performances since we might not use the best threshold for a given exam-
ple and system.
5 Example

5.1 TreeAdd

The parallel version of treeAdd should benefit from a good chunking model. With a simple tree that saves integer numbers in the leafs and no information on nodes, our hybrid chunking strategy will perform a number of optimizations and changes to the code.

```ml
datatype tree = Lf of int |
                  Nd of tree * tree * tree

fun treeAdd (Lf n) = n |
                   treeAdd (Nd (t1, t2, t3)) = let
                               val (x, y, z) = (| treeAdd (t1), treeAdd (t2), treeAdd (t3) |)
                               in
                               x + y + z
                           end

fun mkTree d = let
        fun mk d' = if d' >= d
                        then T.LEAF (1)
                        else T.NODE (| mk (d'+1), mk (d'+1), mk (d'+1) |)
        in
        mk 0
    end
```

The datatype tree represents a binary tree with a leaf that contains an integer value or a node that contains three children of type tree. The `treeAdd` function sums up all integer values of the leafs in the tree. If the tree is big enough it makes sense to split up the task in parallel and let multiple cores compute the values of subtrees.

If `treeAdd` is called with just a leaf then it simply returns its value without any parallel computation. If we have a node, `treeAdd` is called recursively on all children of the node in parallel, hence the hint with the two "||". This parallel recursive call is only efficient if the child nodes have a certain size and therefore the cost of the computation is high enough. Otherwise the overhead of creating the parallel call is much higher than a simple sequential computation of the sum of the nodes.

Our static analysis will change the code to the following representation.

```ml
datatype tree = Lf of int
```
5 Example

(* returns the size of the tree *)
fun sizetree (Lf (_) ) = 1
| sizetree (Nd (size, _, _, _) ) = size

(* computes the cost of the computation with a given tree size as input argument *)
fun treeAddcost ( sizeofinputtree ) =
  if ( sizeofinputtree < 0 )
    then 0
  else c + 3 * treeAddcost ( sizeofinputtree / 3 )

(* the new treeAdd function that switches between parallel and sequential computation depending the the cost of computation *)
fun treeAdd (Lf n ) = n
| treeAdd (Nd (size, t1, t2, t3)) = let
  val (x,y,z) = if ( treeAddcost ( sizetree(t1) ) +
    treeAddcost ( sizetree(t2) ) +
    treeAddcost ( sizetree(t3) ) > Tc )
    then (| treeAdd (t1), treeAdd (t2), treeAdd (t3) |)
  else ( treeAdd (t1), treeAdd (t2), treeAdd (t3) )
  in
    x + y + z
  end

fun mkTree d = let
  fun mk d’ = if d’ >= d
    then T.LEAF (1)
  else let
    val (tree1, tree2, tree3) = (| mk (d’+1), mk (d’+1), mk (d’+1) |)
    val size = sizetree(tree1) + sizetree(tree2) + sizetree(tree3) + 1
    in
      T.NODE(size, tree1, tree2, tree3)
    end
  in
    mk 0
  end

Let us see how we got here.

5.1.1 Static Analysis

In the static analysis we identify the functions that are potential candidates for the hybrid chunking strategy first. Since treeAdd has a PTuple expression and a call to itself in that specific
The next step is to analyze the input arguments of `treeAdd`. The function cost depend on the input argument and therefore we have to take a look at the type of the input argument. It turns out it is of type tree which is the datatype defined on top of the program. This datatype will be compiled in a Cons expression to Construct a tree and therefore we do not have any size information for that type out of the box. We have to rewrite the datatype in the backend to take size information into account. Therefore we will add an additional integer field to the type, the first int field, which will keep track of the size information of each element.

To read out the size information of a certain element we have to create a size function, in the example above called `sizetree`, which returns the integer saved in the first integer field of the type dedicated to the size. This step completes the analysis of the datatype and we can now continue to create the cost function for `TreeAdd`.

We collect all the fix cost in the body of `treeAdd` which are referenced by c in the `treeAddcost` function. Also we encountered 3 calls to `treeAdd` itself in the PTuple expression. The 3 corresponds to the 3 in the body of the cost function \(3 \times treeAddcost(\text{sizeofinputtree}/3)\). Since we assume that the tree is somewhat balanced, meaning that each subtree of a Node has the same size, we can create the cost function with just one call to itself and divide the size by 3.

The next step is to modify the PTuple expression in the body of TreeAdd. We are adding an if statement to the left hand side which determines the cost of the computation inside the PTuple expression. We call the cost function on the size of both input arguments of the call to `treeAdd` in the PTuple expression. This will return us the cost of the computation which we will run in parallel if the cost is bigger than the threshold \(T_c\).

The last step is to modify the mkTree function. We have to create a let statement to collect the size information before creating a node.

5.1.2 Dynamic Decision

We have all the necessary tools in place to make a dynamic decision at runtime. Whenever a tree is created we add size information in form of the number of nodes in the subtree to each of the nodes and one for a leaf. We use these information on the if else clause in the body of the `treeAdd` function to switch between parallel and sequential computation.

5.2 Listsplit

This example takes a list and splits it up in parallel until each sublist just contains one element. Listsplit is an interesting example because it is part of merge sort. The original program looks like this.
5 Example

```plaintext
datatype list = []
    | int * list

fun createintlist (n) = 
  if (n > 0) 
    then 1::createintlist(n-1) 
    else []

fun split ns = let
  (* O(n) algorithm to split the list in half *)
  fun intloop (x::y::xz, xs, ys) = 
      intloop (zs, x::xs), y::ys)
  | intloop (x::[], xs, ys) = (x::xs, ys)
  | intloop ([], xs, ys) = (xs, ys)
  in
      intloop (reverse ns, [], [])
  end

fun splitlist (L) = case L
  of [] => []
  | K::[] => L::[]
  | _ => let
      val (list1, list2) = split(L)
      val _ = (| splitlist(list1) , splitlist(list2) |)
  in
      []
  end

We use the standard list datatype for integers in this example. After creating a list with the function createintlist the list will get split up in the splitlist function. We want to see if the hybrid chunking approach can speed up the PTuple expression in the body of splitlist and makes splitting the list faster.

5.2.1 Static Analysis

As in the previous treeAdd example we identify splitlist as a function that has a recursive datatype as input and contains a PTuple expression that could benefit of the hybrid chunking. The first step is to enrich the datatype with size information.

datatype list = []
    | (int * int) * list

fun intlistsizel (intlist) = case intlist
  of [size, _ ]::rest => size
  | [] => 1

26
5 Example

We dynamically generate the corresponding size function `intlistsize` with return size 1 for the empty list case and the saved size information out of the Cons case. The constructor for the intlist has to change as well to save the size information at time of creation.

```ml
fun createintlist (n) = 
  if (n > 0) 
  then let 
    val (x1,x2) = (1,createintlist(n-1)) 
    val size = intlistsize(x2) + 1 
  in 
    [size,x1]::x2 
  end 
  else Nil
```

The limitations of our model force us to predefine the cost for the `split` function. We can not determine which of the three input arguments is the dependent variable that we need to use for the cost function and especially in this case where all of them are of type list we can not dynamically generate a cost function. We manually attach the cost function to `split` before the analysis to make up for the shortfall of our model.

```ml
fun splitlistcost(size) = if (size < 0 ) then 0 else c + 2*splitlistcost(size/2) + c
```

```ml
fun splitlist (L) = case L of [] => [] | K::[] => K::[] | _ => 
  let 
    val (list1, list2) = split(L) 
    val _ = if (splitlistcost(listsize(list1)) + splitlistcost(listsize(list2)) > Tc) 
      then (| splitlist(list1) , splitlist(list2) |) 
      else ( splitlist(list1) , splitlist(list2) ) 
  in 
    [] 
  end
```

We generate the cost function `splitlistcost` with the fix cost `c` that we obtain from analyzing the body of `splitlist`, the PTuple gives us 2 recursive calls to `splitlist` itself which leads to the generation of `2 * splitlistcost(size/2)` and the predefined cost of the split function `c_{split}`.

The last step is to modify the PTuple expression in the body of `splitlist`. We are adding an `if` statement to the left hand side which determines the cost of the computation inside the PTuple expression. We call the cost function on the size of both input arguments of the call to `splitlist` in the PTuple expression. We will run the computation in parallel if the cost is bigger than a certain threshold `Tc`.

27
5 Example

5.3 Compiler

This third example is a simple compiler example. We are taking a very basic language and want to count all free variables, which in our case are all variables in the code.

```datatype
exp =
  If of exp * exp * exp
  | Let of exp * exp * exp
  | Var of int
  | Binop of exp * exp
  | Num of int
```

```fun countfreeVars (exp) = case exp
  of If(exp1, exp2, exp3) => let
      val (x1,x2,x3) = (| countfreeVars(exp1),
                      countfreeVars(exp2), countfreeVars(exp3) |)
    in
      x1 + x2 + x3
    end
  | Let(exp1, exp2, exp3) => let
      val (x1,x2,x3) = (| countfreeVars(exp1),
                      countfreeVars(exp2), countfreeVars(exp3) |)
    in
      x1 + x2 + x3
    end
  | Binop(exp1, exp2) => let
      val (x1,x2) = (| countfreeVars(exp1),
                     countfreeVars(exp2) |)
    in
      x1 + x2
    end
  | Num(num) => 0
  | Var(num) => 1
```

```fun generatestatement (depth) = let
    fun createexp (d) =
      if ( d > 0 )
        then if ( Rand.randFloat(0.0,1.0) < 0.5 )
            then If(Num(Rand.inRangeInt (0, 10)),
                  createexp(d-1) ,createexp(d-1) )
            else Let(Num(Rand.inRangeInt (0, 10)),
                     createexp(d-1) ,createexp(d-1) )
        else if ( Rand.randFloat(0.0,1.0) < 0.5 )
            then Var(1)
            else Binop(Num(Rand.inRangeInt (0, 10)),
                        Var(2))
      in
        createexp(depth)
    end
```
This example is a more complex tree example. We use a function to create a random piece of code that contains if and let expressions with a var or num on the leaves with a depth $d$.

### 5.3.1 Static Analysis

The static analysis adds size information to our datatype `exp` in the following way.

```haskell
datatype exp =
  If of int * exp * exp * exp
  | Let of int * exp * exp * exp
  | Binop of int * exp * exp
  | Var of int
  | Num of int
```

All non leaf expressions contain a integer field for the size information. This leads to the following size function that the backend generates.

```haskell
fun sizeofexp exp = case exp
  of If (size, _, _, _) => size
  | Let (size, _, _, _) => size
  | Binop (size, _, _) => size
  | Var (_) => 1
  | Num (_) => 1
```

The static analysis identifies the `countfreeVars` function as a good candidate for the hybrid chunking since it takes in a recursive data structure and contains multiple PTuple expression with calls to the function itself. The cost function and modified `countfreeVars` look like this.

```haskell
fun countfreeVarsCost(size) = if (size < 0 )
  then 0
  else c + 8*countfreeVarsCost(size/8)

fun countfreeVars (exp) = case exp
  of If(exp1, exp2, exp3) => let
      val (x1,x2,x3) = if (countfreeVarsCost(sizeofexp(exp1))
        + countfreeVarsCost(sizeofexp(exp2))
        + countfreeVarsCost(sizeofexp(exp3)) > Tc)
        then (|countfreeVars(exp1), countfreeVars(exp2),
              countfreeVars(exp3)|)
        else ( countfreeVars(exp1), countfreeVars(exp2),
               countfreeVars(exp3))
      in
        x1 + x2 + x3
      end
  | Let(exp1, exp2, exp3) => let
      val (x1,x2,x3) = if (countfreeVarsCost(sizeofexp(exp1))
        + countfreeVarsCost(sizeofexp(exp2)))
      in
        x1 + x2 + x3
      end
```

29
Our code is underestimating the real cost because it counts all calls in the case statement to the checkfreeVars function. This leads to the split factor of 8 in the cost function.

The generate statement function changes to a representation to account for size information when we create the data structure.

fun generateStatement (depth) = let
  fun createExp (d) = if (d > 0) then let
    val (e1,e2,e3) = (Num(Rand.inRangeInt (0, 10)), createExp(d-1) ,createExp(d-1) )
    val size = size(e1) + size(e2) + size(e3) + 1
  in
    if (Rand.randFloat(0.0,1.0) < 0.5) then If(size, e1,e2,e3)
  else Let(size, e1,e2,e3)
  end
  else if (Rand.randFloat(0.0,1.0) < 0.5) then Var(1)
  else let
    val (e1,e2) = (Num(Rand.inRangeInt (0, 10)), Var(2))
    val size = size(e1) + size(e2) + 1
  in
    Binop(e1,e2)
  end
in createExp(depth) end
5.4 Benchmarks

Our benchmark machine contains 48 AMD cores, provided by AMD Opteron 6174 “Magny Cours” processors [3, 5]. Each processor contains two dies, and each die has six cores. Each six core node (die) has a dual-channel double data rate 3 (DDR3) memory configuration running at 1333 MHz from its private memory controller to its own memory bank. There are two of these nodes in each processor package.

Each core has 64 KB each of instruction and data L1 cache and 512 KB of L2 cache. Each die has 6 MB of L3 cache physically present, but by default 1 MB is reserved to speed up cross-node cache probes.

The machine has a total of 128GB RAM and runs Ubuntu 10.04.4 LTS Server edition with kernel 2.6.32-38-server x86_64 version. More details on the machine and the Numa architecture can be found in our paper [2].

Each benchmark is run 30 times and we report the average of the runs. Each benchmark is run on 4, 8, 16, 32 and 48 cores. The baseline of each chart is our original Manticore implementation and the speedup is plotted on the y axis. Speedup is defined as $\frac{\text{Manticore without Hybrid Chunking}}{\text{Manticore with Hybrid Chunking}}$.

5.4.1 Threshold $T_c$

In Section 3.1 we defined the threshold for the cut off value for cost expressions as $T_c$. Above this threshold we know that we can execute the code in parallel. We want to find one value that works with all programs and not run a separate profiler first that gives us slightly different thresholds for each program. The limitation of this approach is that a fixed threshold might not work as well on all given problems, but we save the time and effort required to profile a new problem and computer hardware before we run code.

We used the treeAdd and compiler examples to find a good threshold and test the robustness of our model.

The Figures 5.1 and 5.2 show the execution time of the treeAdd and compiler examples on a fix number of cores and data sizes with changing threshold. The threshold is plotted on the x-axis and the runtime in seconds on the y-axis, lower runtime is better. The figures show a U shape for the treeAdd example and a L shape for the compiler example. A U shape is what we expect with the correct chunking size. The minimum runtime will correspond to the optimal chunking size.

The L shape suggest that the example is very robust after the threshold of 9000.

We picked the threshold 10000 for the following benchmarks.

Ways to improve the threshold can be a function that determines the threshold at runtime depending on the number of processors or the underlying architecture. One technique to use a
Figure 5.1: Threshold TreeAdd
Figure 5.2: Threshold Compiler
dynamic threshold is autotuning. The goal of an autotuner is to find a value for the threshold that maximizes speed.
Figure 5.3 shows the runtime of the treeAdd example with different tree depths’ from 14 to 17. The different CPUs (4, 8, 16, 32, 48) are plotted on the x-axis and the speedup on the y-axis. Higher speedups are better.

As we can see in the benchmark 5.3 we have up to 20% speedup on lower cores when we have enough data to chunk to keep all processors busy for a longer time. Even with 48 cores we still obtain a speedup of around 10%.

We suspect the odd results on the 32 core benchmark run is due to the Numa architecture of the machine.

5.4.3 Optimized TreeAdd

The last benchmark run is an optimized version of TreeAdd. We hand optimized the code without a compiler transformation. We want to see if we can obtain better results with this strategy. The benchmark setup and baseline are the same than in Figure 5.3. We used the following repre-
5 Example

<table>
<thead>
<tr>
<th>P/Depth</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.2536</td>
<td>1.2498</td>
<td>1.2387</td>
<td>1.2418</td>
</tr>
<tr>
<td>8</td>
<td>1.2339</td>
<td>1.2392</td>
<td>1.2397</td>
<td>1.2430</td>
</tr>
<tr>
<td>16</td>
<td>1.1995</td>
<td>1.2207</td>
<td>1.2347</td>
<td>1.2358</td>
</tr>
<tr>
<td>32</td>
<td>1.1462</td>
<td>1.1008</td>
<td>1.1845</td>
<td>0.7669</td>
</tr>
<tr>
<td>48</td>
<td>1.010</td>
<td>1.0579</td>
<td>1.1123</td>
<td>1.1256</td>
</tr>
</tbody>
</table>

Table 5.1: Speedup TreeAdd

sentation of the \texttt{treeAdd} and \texttt{treeAddcost} function which differ from the implementation.

\begin{verbatim}
(* computes the cost of the computation with a
given tree size as input argument *)
fun treeAddcost ( sizeofinputtree ) = 2* sizeofinputtree

(* the new treeAdd function that switches between parallel
and sequential computation depending the the cost of computation *)
fun treeAdd (Lf n ) = n

| treeAdd (Nd (size, t1, t2, t3)) =
| if ( treeAddcost ( size ) > Tc )
| then let
|   val (x,y,z) = (| treeAdd (t1), treeAdd (t2),
|                   treeAdd (t3) |)
|     in
|       x + y + z
|   end
| else treeAddseq(Nd (size, t1, t2, t3))
|     in
|       x + y + z
| end

and treeAddseq tree = (case tree
| of Lf n => n
| | Nd (size, t1, t2, t3) => let
|   val (x1, x2, x3) = ( treeAddseq t1, treeAddseq t2,
|                       treeAddseq t3 )
|     in
|       x1 + x2 + x3
|   end
| (* end case *))
\end{verbatim}

Figure 5.4 shows the speedup of our optimized treeAdd version. The speedups up to 16 cores are a lot better than our compiler transformed code, above a speedup of three. The behavior is still the same with 32 and 48 cores. The speedup is getting less but it is still above 1.
Figure 5.4: Benchmark TreeAdd Optimized
Table 5.2: Speedup Optimized TreeAdd

<table>
<thead>
<tr>
<th>P/Nodes</th>
<th>5,000,000</th>
<th>15,000,000</th>
<th>43,000,000</th>
<th>129,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.8183</td>
<td>0.8153</td>
<td>0.8170</td>
<td>0.8211</td>
</tr>
<tr>
<td>8</td>
<td>0.8204</td>
<td>0.8200</td>
<td>0.8126</td>
<td>0.8171</td>
</tr>
<tr>
<td>16</td>
<td>0.8320</td>
<td>0.8321</td>
<td>0.8267</td>
<td>0.8334</td>
</tr>
<tr>
<td>32</td>
<td>0.8656</td>
<td>0.9113</td>
<td>0.8689</td>
<td>0.8232</td>
</tr>
<tr>
<td>48</td>
<td>0.9401</td>
<td>1.1737</td>
<td>0.9550</td>
<td>0.7963</td>
</tr>
</tbody>
</table>

Table 5.3: Speedup TreeAdd Unbalanced Ratio 5:3:2

Unbalanced Trees

To show that our assumption that we need balanced trees to make hybrid chunking a good strategy is correct, we ran benchmarks on unbalanced trees. We modified the `mktree` function to the following representation.

```latex
fun mkTree n = let
  fun mk n' = if n' <= 1
    then T.Lf (Rand.inRangeInt (0, 10))
    else T.Nd (| mk (n' x a), mk (n' x b), mk (n' x c) |)
  in mk n
end
```

Instead of the depth d the `mktree` function takes the number of nodes as an argument and creates subtrees with the ratio of a:b:c.

For the first benchmark we used a ratio of 5:3:2 and 5,000,000, 15,000,000, 43,000,000 and 129,000,000 nodes which roughly corresponds to a depth of 14, 15, 16 and 17. Figure 5.5 shows the benchmark of that run. There are just two runs that were faster with chunking turned on, all the others were slower which shows that our assumption of balanced trees is valid.

The second benchmark has a ratio of 8:1:1 and is a lot more unbalanced than the previous example. Figure 5.6 shows slightly better results than 5.5 but still all speedups are below one except the run with 43,000,000 nodes on 32 and 48 cores.
Figure 5.5: Benchmark TreeAdd Unbalanced Ratio 5:3:2

Table 5.4: Speedup TreeAdd Unbalanced Ratio 8:1:1
Figure 5.6: Benchmark TreeAdd Unbalanced Ratio 8:1:1
5 Example

5.4.4 Listsplit

Figure 5.7 shows the runtime of the listsplit example with different lists length (1000, 5000, 10000, 15000 and 20000).

This example shows the limitations of our model. Every time we split a list we have to update the size information in the new list. We basically allocate a new list including size information in the split function. This overhead is bigger than the performance gained by the hybrid chunking strategy. All the benchmark numbers in 5.7 for the chunking strategy have negative speedup, which means the program is slower with chunking turned on.
Table 5.5: Speedup Listsplit

<table>
<thead>
<tr>
<th>P/Depth</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>15000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-0.1626</td>
<td>-0.2134</td>
<td>-0.3218</td>
<td>-0.1475</td>
<td>-0.09565</td>
</tr>
<tr>
<td>8</td>
<td>-0.1538</td>
<td>-0.2118</td>
<td>-0.3193</td>
<td>-0.1450</td>
<td>-0.1114</td>
</tr>
<tr>
<td>16</td>
<td>-0.2486</td>
<td>-0.2114</td>
<td>-0.3055</td>
<td>-0.1002</td>
<td>-0.1106</td>
</tr>
<tr>
<td>32</td>
<td>-0.1026</td>
<td>-0.1964</td>
<td>-0.2295</td>
<td>-0.0145</td>
<td>-0.0982</td>
</tr>
<tr>
<td>48</td>
<td>-0.0053</td>
<td>-0.1802</td>
<td>-0.0584</td>
<td>-0.1317</td>
<td>-0.1264</td>
</tr>
</tbody>
</table>

5.4.5 Compiler

Figure 5.8 shows the runtime of the compiler example with different depth of 20, 24, 25, 26 and 27.

We generate a random program for the compiler example. The generation function takes a depth variable and builds up a program of let, if, var and int statements by random.

The benchmark numbers in Figure 5.8 show speedups up to 16 cores. The 32 and 48 core speedup numbers are below one again. We suspect that there is not enough parallelism in the code to use all 32 or 48 cores to take full advantage of the chunking strategy.

Table 5.6: Speedup Compiler

<table>
<thead>
<tr>
<th>P/Depth</th>
<th>20</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.2464</td>
<td>1.2588</td>
<td>1.2605</td>
<td>1.2577</td>
<td>1.2580</td>
</tr>
<tr>
<td>8</td>
<td>1.2475</td>
<td>1.2471</td>
<td>1.2482</td>
<td>1.2498</td>
<td>1.2535</td>
</tr>
<tr>
<td>16</td>
<td>1.1685</td>
<td>1.1887</td>
<td>1.1948</td>
<td>1.2300</td>
<td>1.2473</td>
</tr>
<tr>
<td>32</td>
<td>0.9474</td>
<td>0.9043</td>
<td>0.9022</td>
<td>0.8612</td>
<td>0.9819</td>
</tr>
<tr>
<td>48</td>
<td>0.9956</td>
<td>0.9271</td>
<td>0.9348</td>
<td>1.3648</td>
<td>0.9785</td>
</tr>
</tbody>
</table>
Figure 5.8: Benchmark Compiler
6 Related Work

Previous work was focused primarily on static analysis, including the cost of computation and calculating the size of data structures.

Hammond was focusing on a type based analysis for inferring size- and cost-equations for recursive, higher order and polymorphic functions. He approaches the problem in two different papers [13] and [16]. Hammond and Loidl [13] use a static analysis of a program to determine cost and size information. They analyze the cost of the expressions as well as the size of data in order to create size and cost functions. To execute a function in parallel they create a "parGlobal" function which uses cost information to dynamically create parallel tasks. Called at runtime, this function determines the parallel execution of the code. In [16] Hammond and Vasconcelos use a more extensive type and effect system to obtain cost and size information for expressions. They are building a type reconstruction and cost inference algorithm which is an extension of Damas-Milner algorithm. The goal is to input an expression into the algorithm which will calculate a sized type, cost effect, a set of constraints and recurrence equations. This information can be used to obtain the cost of the program.

While the work presented in these papers is the most related to our hybrid chunking approach (particularly the cost and size analysis), we extend their contribution in four notable ways. First we use a more sophisticated approach to split the work in parallel chunks. Secondly, we do not extend our functions to take size information as input arguments, we solely rely on the data structures to save that information. Third, our generated cost and size function uses the information about the data structure to compute the cost. Finally, Manticore compiler is more sophisticated in the way that it divides and distributes work within the runtime and scheduler than the "parGlobal" function used in these papers.

Chin, Khoo and Xu [4] concentrate on obtaining size information of recursive data structures. They use a mixed constraint system to obtain size information about sizes in polymorphic types, data in recursive data structures and higher order functions. The mixed constraint system has three components, arithmetic constraints to collect size properties, bag constraints to model the flow of components and term equality constraints for polymorphism [4].

Huelsbergen and Larus [10] [12] researched the topic of dynamic program parallelization in the earlier nineties. At that time there was very little information about dynamic parallelization of code. They used a static and dynamic analysis as well to capture information about code that can be parallelized at runtime. Part of their work focused on how to parallelize memory access and find functions that are side effect free to dynamically run a piece of code in parallel at runtime. There work related to chunking and granularity control focused on list data structures. They used
functions like map to identify the input argument function and list to parallelize the operation if possible.

Xi and Pfenning [17] researched the elimination of array bound checking through dependent types. They concentrated on integers, booleans and lists and extended their type system with size information for these datatypes. They use the information to eliminate array bound checking and list tag checking. Their approach of extending the type system is similar to ours, they attach size information to the datatypes, also they are just concentrating on integers, boolean and lists and not on user defined recursive datatypes.

The logical programming community publishes papers regarding the size and cost analysis of logical programs that influenced our work. Debray, Lopez-Garcia, Hermenegildo and Lin [6] presented a method to obtain a lower cost bound. The main difference is that they obviously try to find a lower bound and not an upper bound for the cost which contains the risk of assigning 0 cost to a function. They try to use the information of a lower bound to guarantee that a parallel execution of a piece of code is not slower than the sequential execution. They are trying to guarantee speedups. To achieve this, they use the size information to execute a part of code in parallel.

Albert, Arenas, Genaim and Puebla [1] concentrate on solving recurrence relations in the cost analysis. They describe a framework that can solve and obtain an upper bound on recurrence relations dynamically generated by a cost model. We choose to focus on a smaller subset of programs for which we know the closed form expression of the programs recurrence relations.

Mera, Lopez-Garcia, Puebla, Carro and Hermenegildo [14] present an approach to determine the execution time of a program for a specific computer platform. They use a static analysis combined with a profiler in their evaluation of a platform. The cost analysis is divided into platform-dependent and -independent parts. They combine cost functions that depend on the size of their input arguments as well as platform-dependent characteristics like allocation time or setting up a function for execution. A profiler is used to collect necessary information and tailor a platform-specific approach. Conversely, in our approach we shift that computation to the Manticore runtime and the scheduler. This alleviates problems related to hardware differences such as scaling with varying numbers of cores or memory size. This allows our analysis to be independent of the underlying platform.
7 Conclusion

The hybrid chunking strategy presented in this paper works on a subset of recursive problems. Due to the overhead we introduce in saving size information to data structures and the extra if statement in the body of recursive functions, a program has run for a longer time to take advantage of the hybrid chunking strategy. The tree examples work very well up to 16 cores and getting closer to the base line with increasing processors afterwards. The listsplit example shows us the limitation of the model. We have to update the size information in each split and therefore the overhead is bigger than the advantage of the hybrid chunking. Also we need data structures that are balanced to take advantage of the hybrid chunking strategy, unbalanced data structure do not run faster.
Bibliography


Bibliography


