Parallel Processing Letters

AUTOMATIC SKELETONS IN TEMPLATE HASKELL

Kevin Hammond*
School of Computer Science, University of St Andrews, North Haugh, St Andrews, UK

Jost Berthold and Rita Loogen*
Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Germany

Received April 2003
Revised July 2003
Communicated by Gaétan Hains and Frédéric Loulergue

ABSTRACT

This paper uses Template Haskell to automatically select appropriate skeleton implementations in the Eden parallel dialect of Haskell. The approach allows implementation parameters to be statically tuned according to architectural cost models based on source analyses. This permits us to target a range of parallel architecture classes from a single source specification. A major advantage of the approach is that cost models are user-definable and can be readily extended to new data or computation structures etc.

1. Introduction

The notion of a *skeleton* or *pattern* capturing computational structure has now become widespread [1]. One common application is to parallel computing, where such patterns can reduce the cost and complexity of producing parallel code, by allowing algorithms to be specified as a conjunction of common patterns of parallel computation. In earlier papers, we have shown how algorithmic skeletons can be written both in the explicitly parallel Eden [2], and in the implicitly parallel GpH [3], developed new functional skeletons corresponding to branch-and-bound and heuristic search [4], and introduced *implementation skeletons* corresponding to implementation-specific instances of high-level skeletons [5]. Such implementation skeletons must, however, be selected by the programmer. This paper considers the problem of selecting appropriate implementations *automatically at compile-time* using new Haskell meta-programming constructs driven by static cost models. The meta-programming approach has the advantage of eliminating dynamic overheads caused by introducing *adaptors* to match high-level specifications to specific implementation skeletons, while still allowing a single source program to be targeted to multiple platforms and architectures. Moreover, implementation skeleton parameters can be automatically tuned at compile-time to suit a target architecture.

*This work is generously supported by EPSRC grants GR/R 70545/01, GR/R 91298/01 and GR/S 15198/01 and by joint travel grants from the British Council/DAAD.

2. Meta-Programming using Template Haskell

Template Haskell [6] extends the non-strict functional language Haskell [7] with features supporting compile-time meta-programming(GHC version 6.0 and later incorporate Template Haskell directly). These extensions allow parts of a program to be automatically generated or manipulated using statically-computed information. The features of Template Haskell most relevant to our work are *splicing*, *quasi-quotation* and *reification*. An expression `expr` that should be evaluated during compile-time is put into the context of a so-called *splice* `$( expr )`. The expression `expr` must be of type `Expr`. Usually, this expression will be the application of some meta-function. Its result *replaces the meta-application at compile-time* to yield the non-meta source program. Templates are defined using the *quasi-quote* notation. Quasi-quote brackets `[|...|]` are placed around Haskell syntax fragments which should not be evaluated during compile-time, but inserted into the expression as they are. The result of a quasi-quotation is of type `Expr`. Finally, *reification* allows the programmer to query the state of the compiler's internal (symbol) tables. The construct `reifyType` can e.g. be applied to an expression to determine its type which can then be inspected by meta-functions. The `reifyDecl` construct is applied to either a type or function, and returns a meta-representation of the type or function declaration. While there is some similarity with the use of automatic *partial evaluation* to identify program fragments that can be evaluated at compile-time, the meta-programming approach has a number of advantages: firstly, it is possible to ensure that code is executed at compile-time even where a partial evaluator cannot positively determine this; secondly, it is possible to generate code based on internal information such as concrete types; and finally, by using source code generators and templates, it is possible to write generic programs, such as *printf*, that could not be typed directly in Haskell. It is only necessary to ensure that the generator and the generated code are type-correct.

3. The Skeleton Framework

To illustrate automatic cost-based skeletonization we show how a standard `map` function could be parallelized using Eden [8]. Eden extends Haskell with syntactic constructs for *explicitly* defining processes, providing direct control over process granularity, data distribution and communication topology [5,8]. Its two main parallel constructs are process abstraction and instantiation. `process::(Trans a, Trans b)=> (a -> b) -> Process a b` embeds functions of type `a->b` into process abstractions of type `Process a b` where the context `(Trans a, Trans b)` states that both `a` and `b` are overloaded values belonging to the `Trans` class of transmissible values. A *process abstraction* `process (\x -> e)` defines the behavior of a process with parameter `x` as input and expression `e` as output. A *process instantiation* uses the predefined infix operator `( # )::(Trans a,Trans b)=> Process a b -> (a -> b)` to provide a process abstraction with actual input parameters. The evalu-

ation of `(process (\ x -> e1)) # e2` dynamically creates a process together with its interconnecting communication channels. The instantiating or *parent process* is responsible for evaluating and sending `e2`, while the new *child process* evaluates the expression (eagerly) `e1[x->e2]` and sends the result to the parent. The *(denotational)* meaning of the expression is that of the ordinary function application `((\ x -> e1)  e2)`. Lists are communicated as streams, i.e. each element is evaluated eagerly by the producer process and then sent automatically to the consumer process. Eden processes are thus encapsulated units of computation: there is no sharing of (lazily evaluated) values between parent and child processes, and consequently, there are no "stray costs" to be accounted for in the parallel cost model [9].

3.1. Farm of processes

In this section we introduce four different Eden skeletons for parallel `map` implementations [2]. The straightforward implementation which creates one process per application of some worker function `f` can be defined as

```
map_par ::  (Trans a,Trans b) => (a->b) -> [a] -> [b]
map_par f = map (( # ) (process f))
```

In general, this will create a large number of processes which may be excessively fine-grained. We can improve this by creating a fixed number of processes, and allocating more-or-less evenly sized sub-maps to each process. The `farm` skeleton implements this scheme, using a static, but configurable, distribution. The main process of the *farm implementation* creates as many processes as there are available processors, distributes the tasks evenly amongst the processes, and collects the results. Each child process applies the worker function to each data item it receives, and returns the results to the parent process. The farm is parameterized on the number of processors `nPEs`, and the distribution and collection functions `unshuffle` and `shuffle`. The `map_par` skeleton creates the required number of processes.

```
map_farm :: (Trans a,Trans b) => (a->b) -> [a] -> [b]
map_farm  = farm nPEs unshuffle shuffle

farm :: (Trans a, Trans b) =>
        Int -> (Int->[a]->[[a]]) -> ([[b]]->[b]) -> (a->b) -> [a] -> [b]
farm nPEs unshuffle shuffle f tasks
     =  shuffle (map_par (map f) (unshuffle nPEs tasks))
```

Different strategies to split the work into the different processes can be used provided that, for every list `xs`, `(shuffle . unshuffle n) xs == xs` holds. A round-robin scheme is considered to give reasonable results in most cases.

3.2. Saving communication cost: self service farm

Using the above `farm` implies that the parent process must supply the input to all child processes in a piecemeal fashion. When the input is already locally available

when each child process is instantiated, then this can instead be supplied as part of the process abstraction itself. In this new skeleton, the process abstraction is constructed dynamically from the input list as a parameter. While this duplicates work, it reduces communication substantially, reducing the input to just one message for process creation.

```
ssf :: Trans b => Int -> (Int->[a]->[[a]]) -> ([[b]]->[b])
          -> (a->b) -> [a] -> [b]
ssf nPEs shuffle unshuffle f tasks
  = shuffle [(worker f ts) # () | ts <- unshuffle nPEs tasks]
    where worker f tasks = process (\() -> map f tasks)
```

3.3. Feedback loop for dynamic load balancing

Our final variant of the basic parallel map involves distributing data dynamically (in the terminology of [10], this is the only true *farm* skeleton). This skeleton, which we call `workpool`, requires as much communication as the first `farm`, but is a much better solution for problems where the complexity of the list elements is irregular. We exploit stream communication in Eden by constructing a feedback loop from each output to some following task.

```
workpool :: (Trans a,Trans b) => Int -> Int -> (a -> b) -> [a] -> [b]
workpool nPEs prefetch f tasks = sortMerge outsChildren
  where outsChildren = [(worker f i) # inputs |
                        (i,inputs) <- zip [0..nPEs-1]
                                          (distribute .. tasks .. requests)]
```

The dynamic data distribution (`distribute`) requires a nondeterministic `merge` operation which combines the outputs in the order in which they are produced. This merged list of outputs is then used to select free processors for each subsequent task, based on which tasks have completed. The final set of results is sorted by a deterministic sorting function (`sortMerge`). While we can expect a much better task distribution (since a complex task on one processor can be balanced by running a series of less complex tasks on other processes), the need to sort the results may introduce considerable overhead.

3.4. *Chunking input and output data*

To increase granularity all four `farm` skeletons allow the input and output data to be processed in chunks of a (configurable) size. Working with coarse-grained macro-tasks instead of fine-grained tasks reduces the communication overhead substantially and thus improves the runtime behaviour on high-latency distributed systems. In subsequent sections, we will show how optimal settings of this parameter can be determined for each skeleton. The following function embeds implementation skeletons into ones with increased task granularity.


```
macro :: Int -> (([a]-> [b]) -> [[a]] -> [[b]])
             -> (a -> b) -> [a] -> [b]
macro size mapscheme f xs
      =  concat (mapscheme (map f) (chunk size xs))
      where  chunk      :: Int -> [a] -> [[a]]
```

Using this `macro` function, we define chunked versions of the previously explained farm skeletons which we will use in the rest of the paper.

```
eden_farm', eden_ssf', eden_workpool' ::
        Integer -> Integer -> (a -> b) -> [a] -> [b]
eden_farm'     nPEs chSize = macro chSize (farm nPEs unshuffle shuffle)
eden_ssf'      nPEs chSize = macro chSize (ssf nPEs unshuffle shuffle)
```

```
eden_workpool’ nPEs chSize = macro chSize (workpool nPEs 2)
```

While the number of nodes and the chunk size remain accessible, other skeleton parameters as the data distribution (`[un]shuffle`) mode and prefetch length (set to two) are fixed now, creating a uniform interface for all farm skeletons.

3.5. Cost models

Cost models have been provided for various Eden skeletons [2], including our farm skeletons, which account for the creation and termination of processes in the critical path, i.e. from initialisation of the main process until all processors are computing in parallel, plus activity from the end of the last child process until main process termination. Two different kinds of parameters are used:

1. *problem-dependent parameters*: size of input `N`, sequential function times `seqTimes`, size of process input/output `sizeIn, sizeOut` and the size of chunks `chunksize`

```
farmCost :: ProblemParams -> SystemParams -> Double
farmCost (Problem N seqTimes sizeIn sizeOut chunkSize)
         (System nPEs latency commStartup commPerWord
                              timeCreateProcess timeStartProcess)
 = timeInit + timeFinal + timeWorker
 where
  timeInit   = nPEs * (timeCreateProcess + time(sizeIn) + timeUnshuffle1)
```

```
             + latency
timeFinal  = latency + time(sizeOut) + timeShuffle1
timeWorker =  timeStartProcess + (N `div` (P*chunksize)) *
              (time(sizeIn) + chunksize * timeF + time(sizeOut))
-- costs of sequential functions
(timeF:timeUnshuffle1:timeShuffle1:_) = seqTimes
-- local CPU costs for sending/receiving
time :: Int -> Double; time n = commStartup + n * chunkSize * commPerWord
```

Figure 1: Cost Model Function for `eden_farm'`

2. *system-dependent parameters*: number of processors `nPEs`, network `latency`, cost to initiate a message `commStartup` and per-word communication cost `commPerWord`, time to create a process on the parent side `timeCreateProcess` and time to start a process on the child side `timeStartProcess`.

Figure 1 shows a Haskell version of the cost model function for `eden_farm'` which corresponds to the definition in [2] except that we have incorporated chunking effects. The cost models for the other skeletons are similar.

4. Automatic Skeletons

Automatic skeletons are meta-functions which generate specialized code based on compile-time *environmental information*, similar to the parameter sets of the skeleton cost models. This information comprises: *system-specific information* such as the machine topology, the number of processors and the latency of the interconnection network; and *problem-specific information* such as the regularity of the problem, its granularity, the costs of evaluating parameters to the skeleton etc. Each automatic skeleton uses a static cost model to drive the choice of a suitable implementation skeleton. This cost model is a meta-function from the system description and a proposed implementation (represented by the meta-level description of the corresponding function declaration) to some cost metric. The cost model is specified as a Haskell library that may be overridden by the programmer. In template Haskell, the type of a cost model, `CostModel` is

```
type CostModel = Environment -> Decl -> Cost
```

By using local or global optimization techniques, such as dynamic programming, over this cost model it is possible to select an optimum implementation for each automatic skeleton. An automatic skeleton is then a template Haskell function from the environmental description to a meta-expression defined in terms of some specific optimization meta-function and cost model. The application programmer can use automatic skeletons simply by replacing a call to some skeleton by the more flexible equivalent automatic skeleton. For example, the `farm` skeleton can be replaced by the `autoFarm` version, `autoFarm :: Environment -> Expr`.

`...(myFarm myParams) f xs ...`	*application program*
$\downarrow$	**programmer or pre-processor replacement**
`...`**$(autoFarm environment)** `f xs ...`	*meta program*
$\downarrow$	**compiler evaluates meta-function applications**
`...`**(bestFarm bestParams)** `f xs ...`	*executed program*

4.1. Representation of the Farm Library

In order to allow implementation skeletons to be generated at compile-time, we need an appropriate internal representation. In this section, we consider a restricted skeleton library `Skeletons` providing the four different parallel farm implementations introduced in Section 3: the naive parallel map (`parmap`), the farm with static task distribution (`farm`), the self-service farm (`ssf`) and the farm with dynamic task distribution (`workpool`). All four implementation skeletons share a common interface which extends the type of the underlying `map` function with two parameters: one giving the number of processor nodes, and a second (chunk size) controlling how the input data is to be clustered. Although we have chosen to use a common type interface here, it is a strength of the Template Haskell approach that we are not restricted to such a choice: provided the generated program is type-correct, we may derive differently-typed code to deal with different situations.

```
module Skeletons where

data FarmName = Parmap | Farm | SSF | Workpool

type FarmType  a b = Integer  ->               -- # nodes
                     Integer  ->               -- chunk size
                     (a -> b) -> [a] -> [b]   -- map interface

type Cost =  Double -- or any ordered type
type SkCostFunction = ProblemParams -> Environment -> Cost

data FarmSkel a b = FarmSkel  FarmName (FarmType a b) SkCostFunction
```

The `FarmSkel` constructor captures the name of the skeleton, information about its type and a skeleton cost function of type `SkCostFunction`. The module provides the skeletons of Section 3, defined as

```
farmSkels = [eden_parmap, eden_farm, eden_ssf, eden_workpool]

eden_parmap   = FarmSkel Parmap   eden_parmap' parmapCost
eden_farm     = FarmSkel Farm     eden_farm'   farmCost
```

```
eden_ssf      = FarmSkel SSF      eden_ssf'    ssfCost
eden_workpool = FarmSkel Workpool eden_wp'     wpCost

parmapCost, farmCost, ssfCost, workpoolCost :: SkCostFunction
```

In the next section we show how to define automatic skeletons in Template Haskell.

4.2. Specification of the environment

In order to demonstrate the principle of architecture-specific skeleton choice, we use an elementary architecture description language. In our example, it describes general and Eden-specific properties of the system which affect the skeleton cost model. Architectural descriptions can be specialized and refined according to the cost model requirements. Here, we show how automatic skeletons work in principle.


```
-- elementary architecture description language
data Environment =
     System Integer       -- no. of processors
            Double        -- network latency
            Double Double -- communication startup time / cost per word
            Double Double -- process creation costs (parent/child)
```

The environmental description includes the number of available processors, network latency, local startup and per-word time for messages between processors as well as the Eden-specific time for the creation of a process on parent and child side. Since the skeleton cost model also uses problem-dependent parameters `Problem` ..., they must either be provided by the programmer or inferred by the compiler. Specializing a program using programmer-provided information or manual cost models falls short of our objective of *automating* the choice of implementation skeleton at compile-time, requiring significant expertise to implement correctly. We thus focus on automated approaches incorporating automatic static cost analyses.

4.3. Cost model integration

The cost model relies on two basic cost functions: a function to estimate function evaluation cost; and a function to infer (maximum) data sizes for input and output. Our example uses a modular design, where the skeletons are provided by a special library. The sequential cost model is provided in a special cost model module.

```
module CostModel where

type Cost =  Double -- or any ordered type
type CostModel = (Environment -> Decl -> Cost,
                  Environment -> Type -> (Integer,Integer))
```

```
eval_cost :: Environment -> Decl -> Cost
data_size :: Environment -> Type -> (Integer,Integer)
```

The cost model will normally be used by the automatic skeleton to generate compile-time cost information based on the characteristics of the application program. The simple model described here uses two cost functions: the `eval_cost` function provides granularity information derived from an expression, whilst the `data_size` function produces information about the size of input and output data structures derived from the type of the process. These two functions are used to determine problem-dependent parameters for the skeleton cost model: size of data structures communicated and complexity of the evaluation. For illustrative purposes, the latter cost result is given as a simple number here: in general, these would be a more complex cost expression. The cost model would also usually incorporate functions to determine the regularity of the problem: for simplicity, these are omitted here.

4.4. Specialize using concrete cost estimation

The results of the cost analysis are combined with a generic cost model for each implementation skeleton (defined in the `Skeleton` module). In Section 3 we showed the cost model for `eden_farm'` which produces an estimate of actual execution cost given the results of the static cost analysis above and a skeleton parameter set. We now introduce a generic minimizing function for every candidate implementation skeleton, which determines execution cost for one skeleton using the best dynamic parameter settings, here: no. of processors and chunk size. Since precise information will not be available until runtime, some cost assumptions must be made if a purely static cost model is to be used. In our example, we assume, for example, that the argument list is long enough to chunk the elements up to some predefined maximum.

```
minimizeSkel :: ProblemParams -> Environment
                  -> FarmSkel -> (Cost, (Integer, Integer))
minimizeSkel (Problem _ timeF sizeI sizeO _ )
             (System nPEs lat cStartup cPerW timeP timeCh)
             (FarmSkel name _ skCF)
                 = findMin ( zip3 (repeat name) costs candidates)
  where costs    = map costFct candidates
        candidates= [(pes,chunk) | pes  <- [1..nPEs],
                                   chunk <- [1,1+stepC..maxC] ]
        costFct   = \(p,c)->
                      skCF (Problem (maxC*nPEs) timeF sizeI sizeO c)
                           (System p lat cStartup cPerW timeP timeCh)
findMin :: Ord b => [(a,b,c)] -> (a,b,c)
```

The auxiliary function `findMin` (not shown) finds the parameter set with the least cost. It is used to select the best implementation skeleton. Selecting the optimal chunk size for the variable part of the cost model is achieved by a brute force trialand-error search. While this may be expensive, the search will be performed only once at compile time, and the cost should therefore be acceptable. The function `autoFarm` can now be defined to evaluate and compare candidates for all skeletons, select the skeleton with minimum cost and generate code to use this skeleton with optimal parameters. This choice is based on both the structure of the definition and on its type[a]

```
autoFarm ::  Environment -> (a->b) -> Expr
autoFarm system f
   = let (name, _, (nn, chs)) = findMin costList
         costList             = map (minimizeSkel problem system) farmSkels
         problem              = Problem 0 timeF sizeI sizeO 0
         (sizeI,sizeO)        = data_size (reifyType f)
         timeF                = eval_cost (reifyDecl f)
     in  (genSkel name nn chs)
   -- skeleton code generator function
```

[a]Note that the implementation of Template Haskell in GHC 6.0 doesn't allow local variables to be passed to `reify` constructs, as with e.g. `reifyDecl f` above; uses of the `reify` constructs will therefore be slightly more complex than suggested here.

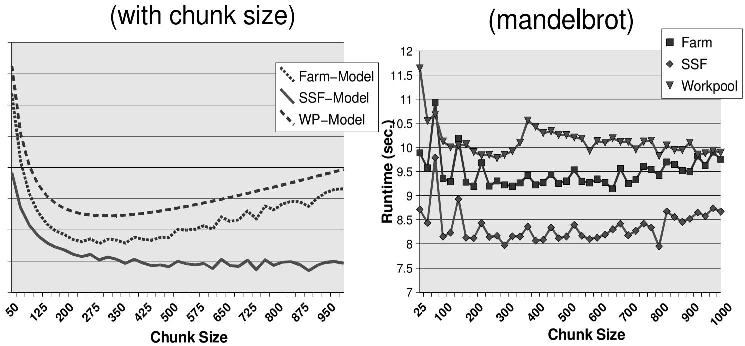

Figure 2: Skeleton cost models and measurements (pixel-wise Mandelbrot)
(8 RedHat 8.0 PCs in a Beowulf Cluster connected by 100MB/s Ethernet)

```
genSkel :: FarmName -> Integer -> Integer -> Expr
genSkel Farm      n  s = [| eden_farm'     $(lift n) $(lift s) |]
genSkel SSF       n  s = [| eden_ssf'      $(lift n) $(lift s) |]
genSkel Workpool  n  s = [| eden_workpool' $(lift n) $(lift s) |]
```

The function `genSkel` uses quasi-quotation to generate the code for the selected skeleton together with its optimal parameters, which are lifted into the generated code. Since both evaluation cost and communication needs are inferred by the cost model plugin from the definition of the worker function, this function must be passed to the `autoFarm`.

```
$(autoFarm environment expensiveFunction) expensiveFunction longList
```

The system will then determine the expected cost (and optimal parameter settings) of evaluating this function for each possible implementation skeleton given the architectural description of the target machine. The most efficient implementation skeleton will then be compiled into the final program.

5. Performance Results using Automatic Skeletons

The impact of different chunk sizes and skeletons as well as the skeleton cost models we use is illustrated by the runtime of a Mandelbrot set visualization using three implementation skeletons from Section 4. Runtime has been measured for the visualization of a heterogeneous 300×300 pixel area on eight processors of a high-latency network (Beowulf Cluster). The measurements show significant differences between different skeletons and an optimal chunk size around 300 pixels. The included cost model curves give a qualitative idea of their behavior. The cost model still has to be refined to get more accurate predictions. Measurements of `farm` and `ssf` show extrema for the chunk sizes 75 and 150, which are due to the problem's heterogeneity, the selected area and its size, while the poor performance of the `workpool` caused by its sorting overhead decreases with big chunk sizes.

6. Related Work

As far as we are aware, this work represents the first attempt to use a template-based approach to skeletal programming in a purely functional language (though Herrmann has also proposed to use Template Haskell in a revised implementation of HDC [11]). Template approaches have been used in other language paradigms, however, most notably C and C++. For example, Ciarpaglini et al's ANACLETO compiler for P3L uses implementation templates which record a good strategy for implementing a given skeleton on a target parallel architecture, supported by a cost model. In contrast to our approach, these templates are provided by the system implementor for each target architecture. The applications programmer is thus unable to modify the templates to deal with new situations. An interesting feature of the ANACLETO approach is the use of profiling information within the compiler to supplement the static cost model by providing information about the execution costs of the sequential parts of the program.

Kuchen's library for C++ [12] exploits the standard C++ template meta-programming system to generate C+MPI implementations from high-level skeletons, delivering good performance compared with more labor-intensive hand-written programs. Meta-programming introduces essential features that are required to implement skeletons effectively: higher-order functions, polymorphism and partial applications. Unlike the approach presented here, however, no attempt is made to use a programmable cost model to drive the choice of implementation; rather the implementation is driven by fixed template instantiation. In Kuchen's approach, different target architectures and applications thus require different template specifications, resulting in either a loss of abstraction or increased implementation effort.

One especially interesting approach is that taken by SkelML [13], which uses *automatic program synthesis* to identify specific parallel patterns. These can then be compiled to yield good parallel implementations. Like most other skeleton approaches, but unlike the work presented here, the cost information that is used to drive the choice of skeleton is embedded as part of the compilation system (as a hybrid static cost analysis/dynamic profiling technique), and is not exposed to the applications programmer. Our approach thus provides the opportunity for the applications programmer to affect the compilation process where this is required, whilst maintaining high-level skeleton abstraction at the source level.

7. Conclusions and Further Work

We have presented a system for automatically deriving parallel implementation skeletons from high-level skeleton specifications in a higher-order non-strict language. Our approach uses meta-programming constructs from the recently implemented Template Haskell system to automatically transform high-level skeletons to good parallel implementations on the basis of static cost information. This cost information is derived from information about the implementation target, the skele-

ton structure and the actual source program using meta-programming constructs. Since the cost model is simply a Template Haskell module whose definitions are used at compile-time, this cost model can be modified by either the applications or systems programmer to deal with new problems, architectural information, user-defined data structures etc. Whilst still at an early stage of development, our results suggest that this should provide a flexible, and hopefully effective, means to write efficient parallel programs.

A number of obvious improvements could be made: firstly we need to extend our work to cover the full range of "standard skeletons"; secondly we should improve the cost models to include concrete performance results for specific Eden implementation skeletons as reported by Rubio et al. [14]; thirdly we should consider the use of hybrid static/dynamic cost models, generating code to obtain required information at runtime; fourthly, we should investigate the exploitation of profiling information at compile-time by calling generated programs from within the Template Haskell meta-program; and finally, we intend to explore compile-time rules for dealing with nested skeletons, as has been done with e.g. SkelML [13]. We would like to thank the anonymous referees for their helpful comments on an earlier draft of this paper.

[1] F. A. Rabhi and S. Gorlatch, eds., *Patterns and Skeletons for Parallel and Dis-*

tributed Computing. Springer, 2002.

[2] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio, “Parallelism Abstractions in Eden,” in *[1]*.

[3] K. Hammond and A. J. Rebon Portillo, “HaskSkel: Algorithmic Skeletons for Haskell,” in *IFL’99*, vol. 1868 of *LNCS*, (Lochem, The Netherlands), Springer, Sept. 1999. *http://www-fp.dcs.st-and.ac.uk/publications/1999/haskskel.ps.gz*.

[4] R. Peña Marí and K. Hammond, “Complex search using eden skeletons.” In Preparation, 2003.

[5] U. Klusik, R. Loogen, S. Priebe, and F. Rubio, “Implementation Skeletons in Eden — Low-Effort Parallel Programming,” in *IFL’00*, vol. 2011 of *LNCS*, (Aachen, Germany), pp. 71–88, Springer, Sept. 2000.

[6] T. Sheard and S. Peyton-Jones, “Template meta-programming for haskell.,” in *Proc. of the workshop on Haskell*, pp. 1–16, ACM, 2002.

[7] S. Peyton Jones and J. e. Hughes, “Haskell 98: A Non-strict, Purely Functional Language,” 1999. *http://www.haskell.org/*.

[8] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña Marí, “The Eden Coordination Model for Distributed Memory Systems,” in *HIPS’97 — Workshop on High-level Parallel Progr. Models*, pp. 120–124, IEEE Comp. Science Press, 1997.

[9] U. Klusik, R. Loogen, and S. Priebe, “Controlling Parallelism and Data Distribution in Eden,” in *SFP’00*, Trends in Functional Programming, pp. 53–64, Intellect, 2000.

[10] M. Cole, *The eSkel Reference Manual*, 2003. *http://www.dcs.ed.ac.uk/home/mic/eSkel/eSkelmanual.ps*.

[11] C. Herrmann, “Using Haskell as a Meta-Language for Skeleton Programming.” *http://www.fmi.uni-passau.de/cl/staff/herrmann/dagstuhl03131/slides.html*.

[12] H. Kuchen, “A Skeleton Library,” *LNCS*, vol. 2400, p. 620ff, 2002.

[13] G. Michaelson, N. Scaife, P. Bristow, and P. King, “Nested Algorithmic Skeletons from

Higher Order Functions," *Parallel Algs. and Applications*, vol. 16, pp. 181–206, 2001.

[14] F. Rubio, *Programación Funcional Paralela Eficiente en Eden*. PhD thesis, Universidad Complutense de Madrid (Spain), November 2001. In Spanish.