Safe Concurrency Introduction through Slicing

Huiqing Li Simon Thompson
School of Computing, University of Kent, UK
h.li@kent.ac.uk s.j.thompson@kent.ac.uk

Abstract

Traditional refactoring is about modifying the structure of existing code without changing its behaviour, but with the aim of making code easier to understand, modify, or reuse. In this paper, we introduce three novel refactorings for retrofitting concurrency to Erlang applications, and demonstrate how the use of program slicing makes the automation of these refactorings possible.

Categories and Subject Descriptors D. Software [*D.2 SOFTWARE ENGINEERING*]: D.2.3 Coding Tools and Techniques

Keywords refactoring; slicing; Erlang; functional programming; concurrency; parallelisation

1. Introduction

Erlang [3, 7] is a functional programming language with built-in support for concurrency based on share-nothing processes and asynchronous message passing. With Erlang, the world is modelled as sets of parallel processes that can interact by exchanging messages. Erlang concurrency is directly supported in the virtual machine, rather than indirectly by operating system threads. Erlang processes are very lightweight, and as a result a program can be made up of thousands or millions of processes that may run on a single processor, a multicore processor or a many-core system.

The advent of the multicore era and the demise of Moore's Law have persuaded programmers to build more parallelism into their programs. However, for the majority of existing Erlang applications, especially those legacy applications written before Erlang's support for symmetric multi-processing (SMP), despite the fact that a certain amount of concurrency is built into the application, the amount of parallelism exhibited is insufficient to keep all the Erlang schedulers as busy as possible. The performance of these applications could therefore be improved by introducing more parallelism to those parts of the application where multi-core resource utilisation is low. Detecting where more parallelism should be introduced to an Erlang application is supported by profiling tools such as Percept2 [20] and etop [2].

The need to retrofit parallelism to existing Erlang applications has given rise to a collection of new Erlang refactorings. Unlike traditional Erlang refactorings, which are mostly structural transformations aiming to make code easier to understand, modify, or reuse, parallelisation-related refactorings are mostly performance-driven; furthermore, some parallelisation refactorings might well make code harder to understand. Another difference we observed between traditional structural refactorings and parallelisation-related refactorings for Erlang is the program analysis techniques needed in order to carry out refactorings; in particular, the use of program slicing is essential for a number of the refactorings that we have automated.

Program slicing is a general technique of program analysis for extracting a part of a program, also called the *slice*, that influences or is influenced by a given point of interest, i.e. the *slicing criterion*. Static program slicing is generally based on program dependency analyses including both control dependency and data dependency. *Backward intra-function slicing*, which extracts the program slice that influences a particular variable/expression, is the kind of slicing used by the implementation of the refactorings proposed.

While there are a number of refactoring tools available for Erlang programs, such as Wrangler [17, 27] and RefactorErl [21], the number of refactorings for retrofitting concurrency is limited. The major contribution of this paper is a set of parallelisation/process-related refactorings for Erlang programs, through which we demonstrate how the use of program slicing techniques makes the automation of these refactorings possible. All these refactorings have been automated, and are supported through the Erlang refactoring tool **Wrangler** developed by the authors.

It has been observed that the refactorings presented here complicate the code, making it more difficult to read and maintain, and that these transformations should be left to a compiler to perform automatically. While this is possible, we prefer the approach presented here for three reasons.

- First, the refactorings can be seen as one component of the general process of program development, and as such they should be an explicit part of the history held in a repository, to be maintained together with other development steps.
- It may well be that an automated approach would not lift precisely the code that a user would wish, and so that the approach presented here may need some manual assistance to "tune" the transformation.
- Complex compiler transformations are notoriously "fragile", with a small syntactic change to a program changing the transformation radically: this is clearly undesirable when the results of the transformation are not visible to the programmer.

The rest of the paper is organised as follows. In Section 2, we give a brief introduction to Erlang and its support for concurrent programming. In Section 3, we give an overview of the existing refactoring and slicing support for Erlang programs. Our slicing-based concurrency introduction refactorings are presented in Section 4, and the facilities used in their implementation are covered in Section 5. In Section 6, we discuss the handling of side-effects in Erlang, as it relates to the work here. Related work is covered in Section 7, and

PEPM '15, January 13–14, 2015, Mumbai, India.

ACM 978-1-4503-3297-2/15/01…$15.00.
http://dx.doi.org/10.1145/2678015.2682533

finally, Section 8 concludes the paper and briefly discusses future work.

2. Introducing Erlang

Erlang [1] is a strict, dynamically typed, functional programming language with support for higher-order functions, pattern matching, concurrency, distribution, fault-tolerance, and dynamic code reloading. Erlang's data types include atoms, numbers and process identifiers, and the compound data types of tuples and lists.

2.1 Syntax: functions, pattern-matching and assignment

The principal definition form in an Erlang module is the *function*. Each function has a fixed number of arguments, its *arity*, and when we need to denote 'the `foo` function with arity 2', we write `foo/2`. The same name can be used for functions of different arities, and these are seen as completely separate definitions: a typical case is where the definition of a function like `foo/2` has an auxiliary function defined by tail recursion, and this would be named `foo/3`.

A function definition consists of a number of *clauses* where each clause consists of a head and a body, separated by an arrow '`->`'; the `qsort` example in Figure 6 has two such clauses. Clauses are separated by semi-colons, '`;`', and the final clause (and so the definition itself) is terminated with a full stop (period), '`.`'.

The *head* of each clause consists of the function name applied to a *pattern* for each argument, (this may be followed by an optional guard, indicated by the keyword `when`). In the `qsort` (Figure 6) the head of the first clause consists of the function applied to an empty list, '`[]`'. The pattern in the second clause, '`[Pivot|Rest]`' matches any *non-empty* list, with `Pivot` matching the first element (or head) of the list, and `Rest` matching the remainder (or tail). Thus, `[...|...]` is the *cons* operation for lists (following Prolog).

When a function is applied to some actual parameters, the *first* clause that matches the parameters is used. As well as variables, patterns can contain the wild-card '`_`', and indeed any variable of the form '`_Foo`' acts as a wild-card, and cannot be used in the body.

The *body* of a function consists of a sequence of statements, separated by commas, '`,`'. These expressions are evaluated in turn, and the result returned by the function is the value of the final expression in the sequence. Expressions can be assignments of the form `Pat = Expr`, where `Pat` is a pattern.

Erlang binding is *single assignment*, so that each (instance of a) variable has a single value. When there is an attempt to re-bind a variable this becomes an attempt to pattern match against a variable that is already bound: this succeeds if the 'new' value is the same as the 'old' and fails otherwise.

Within a module `foo` the function `bar/1` may be called like this

```
bar(Argument)
```

but within another module it must be called in *fully-qualified form:*

```
foo:bar(Argument)
```

(and indeed it may be called in this way within `foo` itself too). In order to understand some of the finer details of the examples, some other notations are seen in Figures 1, 2, 3 and 6.

- Erlang contains records, and these are signalled by the use of '`#`'. In the fragment
  `#child{pid = Pid, ...}`
  the value of the `pid` field of a `child` record is assigned the value (or indeed has its value matched with) the variable `Pid`.
- In line with a number of languages, Erlang has *list comprehensions* that define lists by a combination of generate, test and transform. This is seen in the example
  `[X || X <- Rest, X<Pivot]`
  from Figure 6: this describes the list built by running through the elements `X` of `Rest` and including only those that meet the test `X<Pivot`.
- Finally, Figures 2 and 3 contain an *anonymous function* introduced by `fun`. The function
  `fun(P1 -> B1; P2 -> B2; ...)`
  has exactly the same behaviour as the function
  ```
  anon(P1) -> B1;
  anon(P2) -> B2;
  ...
  ```
  except that is it unnamed. In these examples it is *mapped* along a list by means of the library function `lists:map/2`.

```
-module(echo).

-export([start/0, loop/0]).

start() ->
   Pid = spawn(echo, loop, []),
   Pid ! {self(), hello},
   receive
      {Pid, Msg} ->
           io:format("~p\n", [Msg])
   end,
   Pid ! stop.

loop() ->
   receive
      {From, Msg} ->
          From ! {self(), Msg},
          loop();
      stop ->
         true
   end.
```

Figure 1. A concurrent Erlang program

2.2 Concurrency: processes and message passing

Processes and message passing are fundamental to Erlang. A process is a self-contained unit of computation which executes concurrently with other processes in the system. The primitives `spawn`, '`!`' (send) and `receive` allow a process to create a new process and to communicate with other processes through asynchronous message passing.

The example code in Figure 1 demonstrates process creation, execution and interaction in Erlang. The function `start` initiates a process whose first action is to `spawn` a child process. This child process starts executing the `loop` function from the module `echo` (with an empty list of parameters, `[]`) and is immediately suspended in the `receive` clause waiting for messages of the right format. The parent process, still executing `start`, uses the identifier of the child process (`Pid`) to send the child a message containing a tuple with the parent's process identifier (given by calling `self()`) and the atom `hello`.

Once the message is sent, the parent suspends in the `receive` clause. The child, upon receiving the message from the parent process, sends the message, tupled with its own process identifier, back to the parent. Once the parent has received the echoed message, it prints the message, sends the `stop` message to the child, and terminates. The child receives the `stop`, returns `true`, and terminates.

The main implementation of Erlang is the Erlang/OTP system [1], an open source implementation supported by Ericsson AB. This was first equipped with symmetric multi-processing (SMP) capabilities in 2006, and this support has been improved continuously since then. In the current release (R17), the Erlang Virtual Machine (VM) detects the CPU topology automatically at startup, and creates a scheduler for each CPU core available. Each sched-

uler has it own process run-queue, and processes are migrated between run-queues if scheduler loads need to be balanced [22, 24].

2.3 Erlang/OTP

In addition to the language itself, Erlang comes with the OTP (Open Telecom Platform) middleware library. OTP provides a set of generic behaviours, most notably a *generic server* implementation, together with a *supervisor* behaviour that supports robustness through a hierarchical restart model in the face of component failure.

One of the generic design patterns supported by Erlang/OTP is *gen_server*. A *gen_server* implements a client-server model which is characterized by a central server and an arbitrary number of clients. The server is responsible for managing a common resource shared by different clients. This common resource is represented as the internal state held by the server process.

Two kinds of requests can be sent from client processes to a server process: asynchronous requests and synchronous requests. When an asynchronous request is received, the *gen_server* process only needs to process the request and update its internal state accordingly; no reply needs to be sent back to the client process. On the other hand, when a synchronous request is received, the *gen_server* process needs to calculate two things: the reply which should be sent back to the client – for which the client will wait – and the new value for the state of the *gen_server*.

When implementing a client-server model using Erlang's *gen_server* component, the user needs to define a number of interface functions and callback functions. The callback function for handling synchronous requests must have the following signature:

```
handle_call(Request, From, State)->Result
```

where `Result` is an Erlang tuple. This result typically has the format {`reply, Reply, NewState`}, and uses the Erlang convention that an atom in the first field – here `reply` – identifies or 'tags' the data. The field `Reply` is the value to be sent back to the client, and `NewState` the updated value of the state of the server. The `handle_call` function will typically consist of a number of clauses, each matching a different `Request` pattern in its head.

3. Refactoring and Slicing Support for Erlang

There are a number of refactoring tools for Erlang. **Wrangler** [17, 19] (`https://github.com/RefactoringTools/Wrangler`) is an interactive refactoring and code inspection tool for Erlang developed by the authors. It is implemented in Erlang, and integrated with (X)Emacs and with Eclipse. One of the features that distinguish Wrangler from most other refactoring tools is its user-extensibility. Wrangler provides a high-level *template- and rule-based API* [17], so that users can write their own refactorings, or general program transformations, in a concise and intuitive way without having to understand the underlying Abstract Syntax Tree (AST) representation and other implementation details. User-defined refactorings can be invoked via the Emacs interface to Wrangler, in exactly the same way as built-in refactorings, and so their results can be previewed and undone. Wrangler also provides a domain-specific language (DSL) for composing large-scale refactorings from elementary refactorings.

Complementing Wrangler, RefactorErl [21] is another interactive refactoring tool for Erlang. RefactorErl takes a database approach to store the syntactical and semantical information of the application under refactoring. More recent developments to RefactorErl concentrate on its facilities for program analysis rather than transformation [28].

ParTE [5] is a new refactoring tool built on top of Wrangler and RefactorErl. In particular, Wrangler's API and DSL support for scripting is used in ParTE to build refactorings [6], whereas RefactorErl's program analysis support is used to find parallelisable code candidates. The approach used by ParTE to introducing concurrency is to use an abstract skeleton library called Skel (`skel.weebly.com`). Skel is a collection of common patterns of parallelism that hide explicit process manipulation behind the scene.

In the area of program slicing for Erlang programs, M. Tóth *et al.* [4, 29] have investigated the use of data, behaviour and control dependency information to carry out inter-function forward slicing. Their aim was to detect the impact of a change on a certain point of the program so as to reduce the number of regression test cases to be rerun after the change. In [26], J. Silva *et al.* investigated the use of a system dependence graph (SDG) to support inter-function backward slicing of sequential Erlang programs.

Comparing with program slicing for imperative programs, program slicing for functional programs has its own peculiarities. For instance, Erlang does not contain loop commands such as `while`, `for` or `repeat`. All loops are implemented through recursion. In Erlang, variables can only be assigned once, and pattern matching is used to control the execution flow of a function.

Intra-function backward slicing is supported by Wrangler, and used by the implementation of the refactorings to be introduced in this paper. Since the slicing is within the scope of a function clause, only control and data dependency are used. Instead of generating a dependency graph for programming slicing purposes, Wrangler uses the AST annotated with extra semantic and dependency information as the internal program representation. The advantage of using an annotated AST is that we have a single internal representation for both program slicing and refactoring.

4. Slicing-based Concurrency Refactorings

In this section, we propose three slicing-based refactorings for introducing concurrency to Erlang applications. We would like to point out that there are many other ways for introducing concurrency to an Erlang application. For instance, the use of the sequential *map* operation over a list of data can be refactored to use parallel *map* instead; server processes can be replicated to handle client requests, and so on. In this paper, we focus only on those new refactorings in which the use of slicing plays an important role. We explain these refactorings one by one in more detail now.

4.1 Spawning a worker process for `handle_call`

As we noted earlier, one of the generic design patterns supported by Erlang/OTP is *gen_server*, and two kinds of request can be sent from client processes to a server process: asynchronous requests and synchronous requests. In this section we show how synchronous requests can be transformed to asynchronous ones under certain circumstances.

Requests sent to a *gen_server* process are handled sequentially. Depending on the amount of computation a *gen_server* needs to do when handling a request, there can be situations when a *gen_server* process is overloaded with request messages. Hence it is good practice to check the clauses of the `handle_call` function and see whether any of them can be divided into two parts: one that must be executed on the main *gen_server* process because it affects the state, and the another that does not affect the server state and may be executed in a worker process spawned for it. For instance, the `handle_call` clause shown in Figure 2 can be refactored to that in Figure 3 using our tool Wrangler.

In the code after refactoring, a new worker process is spawned, using `spawn_link`, to carry out the computation of `Resp`. The result is then sent back to the client through a different mechanism, namely:

```
handle_call(which_children, _From, State) ->
     Resp = lists:map(fun(#child{pid = ?restarting(_), name = Name,
                                 child_type = ChildType, modules = Mods}) ->
                              {Name, restarting, ChildType, Mods};
                         (#child{pid = Pid, name = Name,
                                 child_type = ChildType, modules = Mods}) ->
                              {Name, Pid, ChildType, Mods}
                      end,
                      State#state.children),
     {reply, Resp, State};
```

Figure 2. Introduce a worker process to `handle_call` (code before refactoring).

```
handle_call(which_children, From, State) ->
    spawn_link(
      fun () ->
              Resp =
                      lists:map(fun(#child{pid = ?restarting(_), name = Name,
                                           child_type = ChildType, modules = Mods}) ->
                                        {Name, restarting, ChildType, Mods};
                                   (#child{pid = Pid, name = Name,
                                           child_type = ChildType, modules = Mods}) ->
                                        {Name, Pid, ChildType, Mods}
                                end,
                                State#state.children),
               gen_server:reply(From, Resp)
      end),
    {no_reply, State};
```

Figure 3. Introduce a worker process to `handle_call` (code after refactoring).

```
  gen_server:reply(From, Res)
```

Note also that the pattern match in the head of the `handle_call` has changed, since in this case the value of this parameter is used in the body of the function, unlike the case before the refactoring.

The *gen_server* process itself does not wait for the child process to finish, instead it returns immediately from this `handle_call` function with the return value of {`no_reply, State`}. Thus it can be seen that that the two components of the {`reply,...,...`} are returned by the transformed code, but in this case by two separate mechanisms.

In the particular example showing Figures 2 and 3 the state was not modified by the `handle_call` function; it is possible to perform the refactoring even in the case that the state *is* modified, so long as this calculation can be separated from the calculation of the reply. Once the new state computation is complete, the *gen_server* can process the next message: note that here we're performing parallelisation within the processing of a single message; we are *not* parallelising the implementation of the *gen_server* itself, which would be substantially more of a challenge.

This refactoring makes the assumption that the last expression of the `handle_call` function clause has a particular format, namely {`reply, Reply, NewState`}, where `Reply` and `NewState` are both variables. If that isn't the case, it is straightforward to refactor the code so that it has this format before invoking the transformation.

Slicing is used by this refactoring to decide which part of the computation of a `handle_call` function clause can be moved to a new process. We will use S_R and S_N to represent the program slices regarding the slicing criteria `Reply` and `NewState` respectively. Both S_R and S_N consist of a list of top-level expressions from the function clause body, and those expressions do not have to be contiguous. The expressions that can be computed in a worker process are those included in the set difference $S_R \setminus S_N$. The spawning of the new process is placed just before the last expression of the `handle_call` clause.

4.2 Introduce a new process

The refactoring ***Spawn a worker process for*** `handle_call` handles a special case of introducing concurrency, as it can only be applied to a `handle_call` function defined in a `gen_server` implementation. A more general case is to spawn a new process to execute a task in parallel with its parent process, with the computation result of the new process being sent back to the parent process, which will consume it.

As an example, Figure 4 shows a function that sequentially performs image processing on data from two files; Figure 5 shows the result of refactoring to introduce a new process to calculate the values of $R1$ and $F1$ of the code from Figure 4. The highlighted code in Figure 4 is the program slice of the slicing criterion selected, that is the expression sequence: R_1, F_1. In order not to block the execution of the parent process, the `receive` expression is placed immediately before the point where the result is needed, 'as late as possible' in the computation.

If the task to be executed by the new process consists of a sequence of contiguous expressions, the user could just highlight this block of expressions, and apply the refactoring. However this refactoring process could also be driven by the target of the task, in which case a user may want to move into a new process only those parts of the computation that influence the value of the target. In this case, a backward slice of the target could reveal the code fragments, which may or may not be contiguous, that influence the value of the target expression. In order to move those parts of the slice that only influence the target selected, i.e. have no influence on any other code, a static analysis of the annotated AST presentation of the slice is then carried out to remove those expressions that have influence beyond the slicing criterion selected. We place the `spawn_link` expression right after the last expression on which the

```
readImage(FileName, FileName2) ->
   {ok, #erl_image{format=F1, pixmaps=[PM1]}}
      = erl_img:load(FileName),
   Cols1 =PM1#erl_pixmap.pixels,

   {ok, #erl_image{format=F2, pixmaps=[PM2]}}
      = erl_img:load(FileName2),
   Cols2=PM2#erl_pixmap.pixels,

   R1 = [B1||{_A1, B1}<-Cols1],
   R2 = [B2||{_A2, B2}<-Cols2],

   {R1, F1, R2, F2}.
```

Figure 4. Introduce a new process (before)

```
readImage(FileName, FileName2) ->
  Self = self(),
  Pid = spawn_link(
    fun () ->
      {ok, #erl_image{format=F1,
                      pixmaps=[PM1]}}
         = erl_img:load(FileName),
      Cols1 =PM1#erl_pixmap.pixels,
      R1 =[B1||{_A1, B1}<-Cols],
      Self ! {self(), {R1, F1}}
    end),

  {ok, #erl_image{format=F2, pixmaps=[PM2]}}
     = erl_img:load(FileName2),
  Cols2=PM2#erl_pixmap.pixels,

  R2 = [B2||{_A2, B2}<-Cols2],

  receive {Pid, {R1, F1}} -> {R1, F1} end,

  {R1, F1, R2, F2}.
```

Figure 5. Introduce a New Process (after)

```
qsort([]) -> [];
qsort([Pivot|Rest]) ->
  qsort([X || X <- Rest, X<Pivot]) ++
  [Pivot] ++
  qsort([X||X<-Rest, X>=Pivot]).
```

Figure 6. Quicksort in Erlang

slice has a dependency, and the `receive` expression immediately before the expression where the result slice is used so that there is the maximum computation time for the new process to complete before its result is needed.

4.3 Parallelise tail-recursive functions

Iteration, or looping, in functional languages is in general implemented via *recursion*. Recursive functions invoke themselves, allowing an operation to be performed repeatedly until a *base case* is reached. For example, Fig 6 shows a recursive implementation of *quicksort*. A parallel version of this quicksort function can be implemented as shown in Figure 7. In order to control the granularity of parallelism, a new parameter `P` is added, which specifies the maximum number of processes that can be spawned. The value of `P` is generally decided by the number of cores available on the machine. Granularity control is especially useful for parallel recursive functions, to avoid spawning too many processes. Note that the

```
par_qsort(List) -> par_sort(P, List).

par_qsort(0, List) -> qsort(List);
par_qsort(P, []) -> [];
par_qsort(P, [Pivot|Rest]) ->
  Parent = self(),
  spawn_link(fun() ->
    Parent ! par_qsort(P-1, [X || X<-Rest, X>=Pivot])
  end),
  par_qsort(P-1, [X||X<-Rest, X<Pivot]) ++
  [Pivot] ++
  receive Result -> Result end.
```

Figure 7. Parallel Quicksort in Erlang

```
fac(N) -> fac(N, 1).

fac(0, Acc) -> Acc;
fac(N, Acc) when N>0 -> fac(N-1, N*Acc).
```

Figure 8. Tail-recursive `factorial`

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};

do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

Figure 9. An example tail-recursive list processing function

sequential definition of `qsort` is still needed even with the parallel version.

The *qsort* example represents one style of writing recursive functions, i.e. *general recursion*, where the recursive call to itself can happen anywhere in the function body. Another style of writing recursive functions is called *tail recursion*. A recursive function is tail-recursive if the recursive call is the last thing the function does (before returning). Tail-recursive functions often use an accumulating parameter to hold the partial results of the calculation. As an example, Figure 8 shows a tail-recursive implementation of the `factorial` function. Two functions are defined in this example, namely `fac` with the arities of `1` and `2`, since in Erlang different arities mean different functions. The parameter `Acc` to the second function is the accumulating parameter, which holds the result of the function as it is calculated.

While some tail-recursive list processing functions can be automatically refactored to an explicit *map*, or *map-reduce* operation, many are not straightforward without knowledge of the domain. For instance, the example shown in Figure 9 does a recursion over the list `Nodes` while accumulating results to the accumulator variable `Acc`. Each recursive call processes a number of elements in `Nodes`, and the values of `NumGroup` and `Counter` depend on their values in the previous recursion. The recursion reaches its base case when either the list `Nodes` becomes empty or the value of `NumGroup` becomes 1.

Suppose the computation of `make_group(Group, Counter)` is expensive, and there is a need for performance improvement, then simply spawning a new process to do this computation, as shown in the previous examples in Figure 5 and Figure 7 would not help, as the result returned by this computation is immediately

needed by the next expression. Spawning a new process in this case will immediately put the current process into a waiting state. In order to handle this kind of situation, we examined a set of direct tail-recursive functions that meet certain constraints, and developed a new refactoring, ***parallelise tail-recursive function***, for automating the parallelisation of such functions.

The rationale behind this refactoring is to identify the computation component that is independently repeated in every recursion, and delegate the computation task to a worker process so that the main recursion can be run in parallel with a number of worker processes. The pre-condition analysis and transformation of this refactoring are described in more detail in what follows.

This refactoring takes a function definition as input, and carries out a sequence of static analyses to decide whether the function meets the pre-conditions of the automatic parallelisation refactoring. These steps are:

Step 1 is to check whether the function is a direct tail-recursive function with one or more base case clauses, and the only recursive calls appear as the last expression of the function clause body. For simplicity, in this paper we assume the tail-recursive function is of the following form:

```
fun_name(Arg_11, . . ., Arg_1n) -> Body1;
. . .
fun_name(Arg_m1, ...., Arg_mn) ->
   BodyExpr1,
   BodyExpr2,
   . . .
   fun_name(NewArg_m1, ..., NewArg_mn).
```

where the last function clause is the recursive function clause, and all the other function clauses handle base cases. The approach described here can equally well be applied to tail-recursive functions with multiple recursive clauses, and so the assumption here is without loss of generality.

Step 2 is to distinguish accumulating parameters from non-accumulating parameters. In this step, data dependency is used as a heuristic to decide whether or not a parameter is an accumulating parameter. We assume that a parameter is an accumulating parameter if

- its value is influenced by its own value and (possibly) the value of some other parameters,
- the parameter itself does not influence the value of any other parameters, and
- the value of the parameter is not used as a base case condition to terminate the recursion.

Program slicing is used to decide the dependency between parameters in the recursive function clause. In particular, each argument to the recursive function call, i.e. `NewArg_mi`, is selected as a slicing criterion, and its backward slice is calculated. We say that argument `Arg_mi` depends on argument `Arg_mj` if `Arg_mj` is included in the backward slice of `NewArg_mi`.

Taking the `do_grouping` example shown in Figure 9 as an example, the program slices for some of the recursive call arguments are shown in Figure 10. From the slicing result, together with further analysis of the base cases, we are able to conclude that `Acc` is the accumulating parameter of `do_grouping`.

Step 3 is to partition the recursive function clause body. Once the accumulating parameter has been identified, the refactoring needs to decide which part of the computation should be delegated to worker processes, and which part should stay in the main loop. With our approach, the part of the computation that can be moved to a worker process is extracted from the program slice of the

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

(a) program slice for `Remain`

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

(b) program slice for `NumGroup-1`

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

(c) program slice for `NewAcc`

Figure 10. Program slices

accumulating parameter. To be more precise, only the part of the slice that does not depend on the value of the accumulator itself, and does not overlap with the slices of other parameters is to be moved to the worker process.
With the `go_grouping` example, the piece of computation that can be delegated to worker processes is:

```
NewGroup=make_group(Group, Counter).
```

So, up to this point, this refactoring will proceed only if the code fragment that can be moved to a worker process is not empty. Of course, the user could also abort the refactoring process if s/he thinks that the computation to be delegated to worker processes is not the critical part of the computation.

The remaining clause body is further partitioned into two parts. A part for evaluating the new values of recursion control parameters, and a part for evaluating the new value of the accumulating parameter. The former is executed before a task is dispatched to a worker process, and the latter is executed after a result has been received from a worker.

To illustrate how the transformation part of this refactoring works, we continue with the `do_grouping` example. The parallelised version of `do_grouping` resulting from this refactoring is shown in Fig 11. As this example shows, the original tail-recursive function is replaced with a non-recursive function with the same interface. The function `do_grouping` starts by spawning a number of worker processes executing the function `do_grouping_worker_loop/1`. The number of worker processes to be spawned is the same as the number of schedulers available on the Erlang VM. After that, another process is spawned by the entry function `do_grouping_dispatch_and_collect_loop/5`. As its name indicates, this function is in charge of dispatching new tasks to worker processes, collecting results from worker processes in a specific order, and handling the base cases. We refer to this as the *dispatch and collect* process.

```
1. do_grouping(Nodes, Size, NumGroup,Counter, Acc) ->
2.   Parent = self(),
3.   Workers = [spawn(fun() ->
4.                       do_grouping_worker_loop(Parent)
5.                    end)
6.              || _ <- lists:seq(1, erlang:system_info(schedulers))],
7.   Pid = spawn_link(
8.           fun() ->
9.                do_grouping_dispatch_and_collect_loop(Parent, Acc, Workers, 0, 0)
10.           end),
11.  Pid ! {Nodes, Size, NumGroup, Counter},
12.  receive
13.    {Pid, Acc} ->
14.        [P ! stop || P <- Workers],
15.        Acc
16.  end.
17.
18. do_grouping_dispatch_and_collect_loop(Parent, Acc, Workers, RecvIndex, CurIndex) ->
19.   receive
20.      {[], Size, NumGroup, Counter} when  RecvIndex == CurIndex ->
21.          Parent ! {self(), {ok, Acc}};
22.      {[], Size, NumGroup, Counter} when  RecvIndex < CurIndex ->
23.          self() ! {[], Size, NumGroup, Counter},
24.          do_grouping_dispatch_and_collect_loop(
25.             Parent, Acc, Workers, RecvIndex, CurIndex);
26.      {Nodes,_Size, 1, Counter} when RecvIndex == CurIndex ->
27.           Parent ! {self(), {ok, [make_group(Nodes, Counter)|Acc]}};
28.      {Nodes, Size, 1, Counter} when RecvIndex < CurIndex ->
29.           self() ! {Nodes, Size, 1, Counter},
30.           do_grouping_dispatch_and_collect_loop(
31.             Parent, Acc, Workers, RecvIndex, CurIndex);
32.      {Nodes, Size, NumGroup, Counter} ->
33.           Group = lists:sublist(Nodes, Size),
34.           Remain = lists:subtract(Nodes, Group),
35.           Pid = oneof(Workers),
36.           Pid ! {self(), Group, Size, Counter},
37.           self() ! {Remain, Size, NumGroup-1, Counter+1},
38.           do_grouping_dispatch_and_collect_loop(
39.              Parent, Acc, Workers, RecvIndex, CurIndex+1);
40.       {{worker, _Pid}, RecvIndex, NewGroup} ->
41.           NewAcc = [NewGroup|Acc],
42.           do_grouping_dispatch_and_collect_loop(
43.              Parent, NewAcc, Workers, RecvIndex+1, CurIndex)
44.   end.
45.
46. do_grouping_worker_loop(Parent) ->
47.      receive
48.         {Group, Size, Counter, Index} ->
49.             NewGroup = make_group(Group, Counter),
50.             Parent ! {{worker, self()}, Index, NewGroup},
51.             do_grouping_worker_loop(Parent);
52.         stop ->
53.             ok
54.     end.
55.
56. oneof(Workers) ->
57.     ProcInfo = [{Pid, process_info(Pid, message_queue_len)} || Pid <- Workers],
58.     [{Pid, _}|_] = lists:keysort(2, ProcInfo),
59.    Pid.
```

Figure 11. A parallel implementation of the `do_grouping` function

The initial computing task in then sent to the *dispatch and collect* process as shown in line 11, after that the parent process is suspended in a `receive` clause waiting for the final result to come. Once the final result has been received, the parent process sends a `stop` signal to each worker process to terminate them, then terminates itself and returns the final result to its caller.

On the other hand, the *dispatch and collect* process is immediately suspended in a `receive` clause after its creation. The first message it receives represents the initial computation task sent by the parent process in line 11. This message is then pattern-matched in turn with each pattern in the `receive` expression. If the initial task does not match any of the base cases, then it should match the non-base case clause in line 32. The body of this `receive` clause first executes the program slice (lines 33-34) whose result affects the initial value passed onto the worker process, as well as the next iteration, then selects a worker process from the list of available worker process identifiers: `Pids`.

With the current implementation of this refactoring, the process with the shortest message queue is selected, as defined in the function `oneof` (lines 56-59). Once a worker process has been selected, a message containing the initial parameters for the new task is then sent to the worker process. Instead of waiting for the worker process to return the result, the *dispatch and collect* process continues to run in parallel with the worker process. It sends the remaining task to itself, and iterates the process.

Once a worker process has finished the computation of a task, it sends the result back to the *dispatch and collect* process, and waits for the next message to come. This is defined in the function `do_grouping_worker_loop` in lines 46-54. The *dispatch and collect* process uses indices to track each job dispatched and collected. Results from worker processes are received in the expected order, that is the order in which jobs are dispatched. Once an expected result has been received, this process takes the new result and the current value of the accumulator, calculates and updates the accumulator's value accordingly as shown in lines 40-43. Note that in the tail recursive call in line 43 the `RecvIndex` parameter is incremented: stepping this through by single increments ensures that the results are collected in the same order as the sub-tasks are dispatched, which is crucial for preserving the semantics of the computation.

When a base case message has been received, the process first checks if all the expected results have been received by checking if the two indices `RecvIndex` and `CurIndex` have the same value. If the result is `true`, then the final result is calculated and sent back to the parent process, otherwise the process will have to wait until all the results have been collected. To do so, it sends the base case message to itself so that it will eventually be processed.

As this refactoring shows, the transformation process is rather complex and error prone if done manually. With this refactoring support, the user is able to experiment with the parallel version with little effort.

5. Implementation

Wrangler is a mature refactoring tool for Erlang, written in Erlang, and we have used the facilities of Wrangler to implement the refactorings discussed here. We briefly discuss these here, and refer readers to the articles cited for more information about the details.

5.1 Wrangler in a nutshell

Architecture. Wrangler consists of a pipeline of stages:

- parsing, to give an abstract syntax tree (AST);
- semantic analysis, to give an annotated AST (AAST);
- transformation of the AAST, and
- pretty-printing to file.

and as it presents the top-level functions from each of the stages to build refactorings through the `api_interface`. More details of the architecture can be found in the overview [27].

Syntax. Erlang comes with a `syntax_tools` library that encapsulates various aspects of the syntax including macros; we have extended the tokeniser used by `syntax_tools` to include column information and preserve white spaces and comments.

Analysis. Information from the static semantic analysis is stored in the AAST, and this can be accessed directly or through API functions. These include operations to give the free and bound variables in syntactic components, as well as – for example – allowing the generation of 'fresh' identifiers on demand.

The analyses here are also supplemented with the slicing technology described in the previous section.

Wrangler extensibility. Wrangler supports user extension [18] in two complementary ways.

The **API** [17] allows users to define new refactorings 'from the bottom up'. It provides templates that can describe fragments of (A)AST through fragments of Erlang concrete syntax, augmented with meta-variables that range over syntactic elements. On top of these templates it is possible to build rules explaining how code is to be transformed, when the code meets appropriate pre-conditions. In the work presented here, we use templates in the *code synthesis* needed in building transformed programs.

The **DSL** [19] supports complex scripting of refactorings, with control on their transactional nature, their interactivity, tracking of (renamed) names etc. It can be used in this context to present the facilities in a more exploratory way, allowing uses choices between possible variants of parallelisation refactorings, for instance.

Wrangler is written in Erlang. Functional languages – with pattern matching over structured data, and higher-order functions – are particularly well suited as metalanguages for transformation and analysis, and we leverage that here.

5.2 Implementing Refactorings Using Wrangler

The refactorings presented in this paper are implemented using Wrangler's API, in particular these refactorings implement a behaviour, named *gen_refac*, exposed by Wrangler. In Erlang, a behaviour is an application framework that is parameterised by a *call-back* module. The behaviour implements the generic parts of the solution, while the callback module implements the specific parts. A number of pre-defined behaviours are provided through Erlang/OTP. In the same spirit, the *gen_refac* behaviour implements those parts of a refactoring that are generic to all refactorings, such as the generation and annotation of ASTs, the outputting of refactoring results, the collection of change candidates and the workflow of the refactoring processes. To implement a refactoring using *gen_refac*, the user only needs to implement a number of callback functions, of which the two most important are `check_pre_condition` and `transform`.

To illustrate how these refactorings are implemented, we take the transformation part of *introduce a worker process to handle call* as an example. While the implementation of *tail-recursive function parallelisation* is more complex due to the amount of analysis involved, the methodology used is the same.

The implementation of the callback function `transform` of *introduce a worker process to handle call* is as shown in Fig 12. This function applies a transformation rule defined by function `rule1` to the current file under refactoring. The function `rule1` implements a transformation that modifies a `handle_call` clause at a given position to introduce parallelism. `?RULE` is a macro defined

```
transform(_Args=#args{current_file_name=File,
                      cursor_pos=Pos}) ->
    ?STOP_TD_TP([rule1(Pos)], [File]).


rule1(Pos) ->
   ?RULE(?T("handle_call(Args@@) when Guard@@->
                    Body@@,{reply, Res@, State@};"),
            gen_new_handle_call(_This@, Res@, State@,
                 {Args@@, Guard@@, Body@@, State@}),
            begin
               {S, E} = api_refac:start_end_loc(_This@),
               S=<Pos andalso E>=Pos
            end).

gen_new_handle_call(C, Res, State,
                        {Args, Guard, Body, State}) ->
    {Slice1, _}=wrangler_slice_new:backward_slice(C, Res),
    {Slice2, _}=wrangler_slice_new:backward_slice(C, State),
    ExprLocs = Slice1 -- Slice2,
    Exprs =[B||B<-Body,
              lists:member(
                api_refac:start_end_loc(B), ExprLocs)],
    NewBody = Body -- Exprs,
    api_refac:subst(
        ?T("handle_call(Args@@) when Guard@@ ->
                 Body@@,
                 spawn_link(
                       fun()->
                            Resp= begin Exprs@@ end,
                         gen_server:reply(From, Resp)
                        end),
                 {no_reply, State@};"),
            [{'Args@@', Args}, {'Guard@@', Guard},
              {'Body@@', NewBody}, {'State@', State},
              {'Exprs@@', Exprs}]).
```

Figure 12. Transform a handle_call function clause

in Wrangler used to define transformations. It takes three parameters: a template characterising the program fragment to transform, a description of the new program fragment that replaces the old one, and a pre-condition on the application of the rule; the call takes the form `?RULE(Template, NewCode, Cond)`.

The function `gen_new_handle_call` is the one that generates the new code. This function first calculates the fragment of code to be executed by the new process as well the part that remains in the main process, then generates the new code using a template as indicated by the macro call `?T`.

6. Side-effect Analysis

Being side-effect free plays an important part in functional programming languages. In a side-effect free language, the same expression always produces the same value when evaluated multiple times. This *referential transparency* feature makes program analysis, comprehension and transformation easier. Unlike other functional programming languages such as Haskell and Clean, which have a substantial pure subset, Erlang has controlled side-effects to support communication amongst other features.

Erlang is pure in having immutable data structures and single assignment variables; it is not pure due to its support for concurrency, **E**rlang built-in **T**erm **S**torage (ETS), process dictionary, etc. In an Erlang program, both pure functions and impure functions can be used in any context (except function guards).

For refactorings that do not change the execution context of the code under refactoring – i.e. the process in which the code is executed – there are some workarounds when side-effects are a concern. For example, naively generalising a function over an expression that has side-effects could potentially change the behaviour of the function. The solution to this problem is to wrap the side-effecting expression up as a *closure*, as shown in the *generalisation* example in Fig 13, where the function `print_list` is generalised over the expression `io:format("*")`.

```
print_list(0) -> ok;
print_list(N) ->
  io:format("*");
  print_list(N-1).

test()->print_list(3).
```

(a) Code before *generalisation*

```
print_list(F, 0) -> ok;
print_list(F, N) ->
  F(),
  print_list(F, N-1).

test() ->
  print_list(fun() ->io:format("*") end, 3).
```

(b) Code after *generalisation*

Figure 13. Generalisation over an expression with side-effects

Concurrency-related refactorings are vulnerable to side-effects due to the fact that very often a concurrency-related refactoring needs to migrate some computation from one process to another, and this potentially affects those execution-context-aware operations. For instance, the Erlang built-in function `self()` returns the process identifier of the calling process, hence care has to be taken if a refactoring changes the execution context of `self()`, because in a new process the same expression will return a different process identifier. On the other hand, it might be perfectly ok to migrate some code with side-effects into another process.

In order to support safe concurrency introduction refactorings, side-effect analysis of the code affected by a refactoring is a necessity. With the knowledge that side effects in Erlang are due to a small number of known reasons, but *not* to single assignment, we are able to identify an initial set of functions whose side-effects are predefined. This hard-coded information indicates not only that a function has side-effects, but also specifies the kind of side-effects associated with it.

With this pre-defined side-effect information, static AST-based techniques are then used to establish function dependencies, and side-effect information is then propagated over the dependency graph until a fixed point is reached. To improve the efficiency of side-effect analysis, side-effect information about library functions is pre-computed, and stored in a persistent table.

In the case that the code to be migrated to another process does have side-effects, the user is presented with the side-effect information derived, and it is the user's choice whether or not to continue with the refactoring.

7. Related Work

Slicing. A direct application of program slicing in the field of refactorings is *slice extraction*, which has been formally defined by Ettinger [13] as the extraction of the computation of a set of variables from a program as a reusable program entity, and the update of the original program to reuse the extracted slice. Ettinger's study was not concerned about concurrency.

In [10], J. Cheng extends the notion of slicing for sequential programs to concurrent programs and presents a graph-theoretical approach to slicing concurrent programs. In addition to the usual control and data dependencies, J. Cheng introduces three new types of primary program dependences in concurrent programs, named the selection dependence, synchronisation dependence and communication dependence. The techniques developed aim to help the debugging of concurrent programs by finding all the statements that possibly or actually caused the erroneous behaviour of an execution of a concurrent program where an error occurs.

A more precise slicing algorithm is proposed by J. Krinke: in [15] he proposes a context-sensitive approach to slicing concurrent programs. This approach makes use of a new notation for concurrent programs by extending the control flow graph and program dependence graph to their threaded counterparts.

In [14], M. Kamkar *et al.* propose a tracing-based algorithm for distributed dynamic slicing on parallel and distributed message-passing based applications. With this approach, the authors introduce the notion of Distributed Dynamic Dependence Graph (DDDG) which represents control, data and communication dependences in a distributed program. This graph is built at run-time and used to compute slices of the program through graph traversals.

Parallelisation. There is a substantial literature on parallelisation of programs, the vast majority of which addresses parallel programs in the object-oriented (typically Java, C++) and imperative (x10, Fortran) paradigms. In common with our observation for Erlang, Dig notes in [12] that "unlike sequential refactoring, refactoring for parallelism is likely to make the code more complex, more expensive to maintain, and less portable".

Dig's paper [12] exemplifies the main approach for OO languages in targetting thread-safe libraries and data structures within a general-purpose language, which, once achieved, provides further refactoring opportunities. Alternatively, programs can be targeted at specialised hardware, such as GPUs [11] and multicore systems [9]. These approaches typically require pointer analyses to identify access to mutable data structures, a problem which is not evident in Erlang – which features single assignment – and other functional languages. Working within parallel languages such as x10, which embodies the partitioned global address space (PGAS) model, some work has been done in loop parallelisation [23], and, while these are not included in the main release, there have been some experiments in parallelisation in the Fortran refactoring tool Photran [25]. Other work on loop parallelisation [16] notes the importance of user input into the parallelisation process.

Work on parallelisation of functional programs has typically taken two routes. First, *skeletons* have been used to identify potential sites for parallelisation, and this forms the basis of the work of the ParaPhrase project [6]. Secondly, data parallel systems have been developed – including Data Parallel Haskell [8] – but to the best of our knowledge there has been no work on refactoring for data parallelism in a functional context.

8. Conclusions and Future Work

In this paper, we presented three novel slicing-based refactorings for introducing concurrency to Erlang applications, and in that way parallelising the systems. All these refactorings are automated in the Erlang refactoring tool Wrangler. While there are other ways for retrofitting concurrency to existing Erlang applications, such as the use of skeletons/patterns, our refactorings complement the existing ones. The application of program slicing to the refactoring field is not new, but our work demonstrates its usefulness for supporting concurrency introduction refactorings.

As we noted in the introduction, we have chosen to implement these refactorings explicitly as part of the software development process, rather than implicitly – and 'invisibly' – inside a compiler. We have done this to make the transformation a part of the software development process, and also because we see that fully automated transformations often need some modifications in application or scope in order to deliver precisely what is required. In doing this we agree with others working in the field [16] who note that "automatic parallelization of loops is a fragile process" and so include user input in the process, rather than incorporating the transformation within the internals of a compiler.

Our future work lies in a few directions. First, we will investigate other refactorings, and code inspection functionalities, which can benefit from program slicing techniques; second, we will extend Wrangler to support automatic discovery of candidates where concurrency can be introduced; and finally we will connect Wrangler with concurrency profiling tools such as Percept2 [20] to provide feedback on the performance impact after concurrency introduction.

Acknowledgments

This research is supported by EU FP7 project RELEASE, grant number 287510, (`www.release-project.eu`); we thank our funders and colleagues for their support and collaboration.

References

[1] Erlang/OTP. `http://www.erlang.org`.

[2] ETop - The Erlang Top. `http://www.erlang.org/doc/man/etop.html`.

[3] J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.

[4] I. Bozó and M. Tóth. Building Dependency graph for slicing Erlang Programs. In *Periodca Politechnica*, pages 372–390. 2010.

[5] I. Bozó, V. Fordós, et al. Discovering Parallel Pattern Candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 13–23. ACM, 2014.

[6] C. Brown, K. Hammond, M. Danelutto, P. Kilpatrick, H. Schöner, and T. Breddin. Paraphrasing: Generating Parallel Programs Using Refactoring. In *Formal Methods for Components and Objects*, pages 237–256. Springer, 2013.

[7] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, Inc., 2009.

[8] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM.

[9] F. Chen, H. Yang, W.-C. Chu, and B. Xu. A Program Transformation Framework for Multicore Software Reengineering. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 270–275. IEEE, 2012.

[10] J. Cheng. Slicing Concurrent Programs - A Graph-Theoretical Approach. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, AADEBUG '93, pages 223–240, London, UK, 1993. ISBN 3-540-57417-4.

[11] K. Damevski and M. Muralimanohar. A Refactoring Tool to Extract GPU Kernels. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, pages 29–32, New York, NY, USA, 2011. ACM.

[12] D. Dig. A Refactoring Approach to Parallelism. *IEEE Software*, 28 (1):17–22, 2011.

[13] R. Ettinger. Refactoring via Program Slicing and Sliding. In *23rd IEEE International Conference on Software Maintenance*, pages 505–506, Paris, France, 2007.

[14] M. Kamkar, P. Krajina, and P. Fritzson. Dynamic Slicing of Parallel Message-Passing Programs. In *PDP*, pages 170–178. IEEE Computer Society, 1996.

[15] J. Krinke. Context-sensitive Slicing of Concurrent Programs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 178–187, New York, NY, USA, 2003.

[16] P. Larsen, R. Ladelsky, J. Lidman, S. McKee, S. Karlsson, and A. Zaks. Parallelizing more Loops with Compiler Guided Refactoring. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 410–419. IEEE, 2012.

[17] H. Li and S. Thompson. A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, School of Computing, Univ. of Kent, UK, 2011.

[18] H. Li and S. Thompson. Let's Make Refactoring Tools User-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 32–39. ACM, 2012.

[19] H. Li and S. Thompson. A Domain-Specific Language for Scripting Refactorings in Erlang. In *15th International Conference on Fundamental Approaches to Software Engineering(FASE)*, pages 501–515, 2012.

[20] H. Li and S. Thompson. Multicore profiling for Erlang programs using percept2. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, pages 33–42. ACM, 2013.

[21] L. Lövei, C. Hoch, H. Köllő, T. Nagy, A. Nagyné-Víg, D. Horpácsi, R. Kitlei, and R. Király. Refactoring Module Structure. In *Proceedings of the 7th ACM SIGPLAN workshop on Erlang*, pages 83–89, Victoria, British Columbia, Canada, Sep 2008.

[22] K. Lundin. About Erlang/OTP and Multi-core Performance in Particular. Erlang Factory London 2009.

[23] S. A. Markstrum, R. M. Fuhrer, and T. D. Millstein. Towards Concurrency Refactoring for x10. *SIGPLAN Not.*, 44(4):303–304, 2009.

[24] P. Nyblom. Erlang SMP Support. Erlang User Conference 2009.

[25] J. Overbey, S. Xanthos, R. Johnson, and B. Foote. Refactorings for Fortran and High-performance Computing. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 37–39, New York, NY, USA, 2005. ACM.

[26] J. Silva, S. Tamarit, and C. Tomas. System Dependence Graphs in Sequential Erlang. In J. de Lara and A. Zisman, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 486–500. Springer Berlin Heidelberg, 2012.

[27] S. Thompson and H. Li. Refactoring tools for functional languages. *Journal of Functional Programming*, 23(03):293–350, 2013.

[28] M. Tóth and I. Bozó. Static Analysis of Complex Software Systems Implemented in Erlang. In V. Zsák, Z. Horváth, and R. Plasmeijer, editors, *Central European Functional Programming School*, volume 7241, pages 440–498. 2012.

[29] M. Tóth, I. Bozó, et al. Impact Analysis of Erlang Programs Using Behaviour Dependency Graphs. In Z. Horváth et al., editors, *Central European Functional Programming School*, Lecture Notes in Computer Science, pages 372–390. 2013.