Function Flow Project

March 31, 2013

Abstract

`Function Flow` aims to explore compilers' opportunites to task parallelism. We believe current programming languages and compilers should and can be more proactive in improving productivness of parallel programming. `Function Flow` achieve this by

- providing a new language extension which unify expressing parallelsim,
- static analyzer to report concurrency bugs,
- automatic optimizer for task parallelism,
- runtime which can leverage parallelism information.

1 Motivation of **`Function Flow`**?

As we claimed, our target of `Function Flow` is to exploit compilers' abilities of improving productivness in parallel programming. Then what are the main aspects of productivness decreasing in parallel programming? In `Function Flow`, we class them with the following orthogonal aspects and try to fill the gap.

- Difficult to express parallelism. Expressing parallelism is not only about how to parallelize specific code, but also about parallel tasks' orders, or dependences. We find that they, parallelization and coordinating parallelism, are so close that decoupling them is anti-intuitive. In `Function Flow`, one can coordinate parallel tasks at the time creating them without any modification of existed business logic code.
- Difficult to debug. There are many kinds of concurrency bugs in parallel applications, such as deadlock, data race, etc. But one of the most important reason of debugging is concurrency bugs are hard to reproduce. `Function Flow` tries to report possible bugs at compile time, but not runtime. `Function Flow` achieves this by analyze static dependency graph (SDG) extracted from `Function Flow` language. As SDG contains orders of parallel tasks, it's possible to identify possible bugs more precisely in `Function Flow`.

```
spawn{
  para b, c, d, e, f;
  for(...)
      para A();
  b B();
  c C();
  e<b && c> E();
 }end(any(anony) || b){
   d D();
   f<d && e> F();
}
```

Figure 1: An implementation from `Function Flow`.

- Difficult to optimize. Until now, researchers are still trying to identify factors which affect parallel programs performance, such as cache miss rate, sharing cache, fasle sharing, scheduling in operating system, conflicts on locks or shared data structure, and etc. It's even harder to leverage corresponding optimization opportunites than identifying them. `Function Flow` exploits several optimizing techniques and more ideas are waiting for our exploration.

2 Function Flow Language

Techniqually, `Function Flow` can be integrated with C/C++, Java, C#, and etc. Currently, we have integrated it with C, and C++ is still under heavy development.

In `Function Flow`, a basic parallel units is a function. Functions are declared to be parallel tasks by using a named higher-order functor `para`. Then a task is spawned with a condition that acts as its execution prerequisite. As it denotes the termination of other tasks, it can express task dependency without decreasing parallelism.

As an illustration, here is a example in Fig. 1. Here, `para` is a higher-order functor and is used to create a new task either using a named functor as with `b`, to call function `B`, or an anonymous functor, such as that used to call `A`. If a functor is named, then the task created by it can be referred to by the name of the functor. Each `para` functor has two parameters. The first parameter, between angle brackets `<...>`, is the pre-requisite for executing the task, in terms of the terminations of other tasks. We call this a functional wait. The second parameter is the procedure to be parallelised. So a `para` functor spawns the specified task by calling the specified procedure.

More details here!

3 Static Dependency Graph (SDG)

The static dependency graph enables us to infer execution sequences at compile time and thereby detect concurrency bugs and potential optimisations. It

is built with the following steps:

- building SSA (Static Single Assignment) for `para` and `cond` variables via the following definition, this step generates representation α;

  Definition 1 *If `a` waits for `b`, then `a` is defined by `b` while `b` is used by `a`.*

- building data dependences and control flow dependences for `para` tasks on representation α, named β;
- building call graph, CG, for methods which take `para` or `cond` as parameters, or take `para` or `cond` as return values;
- connecting call graph CG and dependences graph β, this step generates **SDG**.

`Function Flow` can precisely build SDG. This is largely because static analysis in `Function Flow` is simpler than in traditional program analysis. Fistly, `cond` and `para` variables are declared and used with the original type in `Function Flow`. And reference or pointers to `cond` and `para`, are not allowed in `Function Flow`. Secondly, variables other than `para` and `cond` are not considered. The only exception is if a control flow can reach a `para` or `cond` variables' definition, the control flow shall be analyzed to check if it depends on some `para` task's return value. Such as the following example shows.

```
bool flag = para DoSomething(...);
if(flag)
  para DoAnother(...);
```

Here, the `flag` shall be checked to see if it depends on a `para` task. Thirdly, call graph only contains functions with `para` or `cond` either as parameters or return values. These largely reduces the complexity of intra-procedure analysis in `Function Flow` and makes it affordable. Here is an example of SDG.

4 Cases

We have two cases (1, 2) to illustrate how `Function Flow` implements and reports concurrency bugs and optimizing opportunites.

4.1 Dedup in PARSEC

`Dedup`, from PARSEC [**?**], is a routine that compresses data with a "next-generation" method called deduplication, now much used for backup storage and network transmission of large amounts of data, with a pipelined architecture as follows:

- break the input stream into small chunks;
- for each chunk, compute a hash value that uniquely identifies its content;
- insert it into a global database of chunks indexed by the hash value;

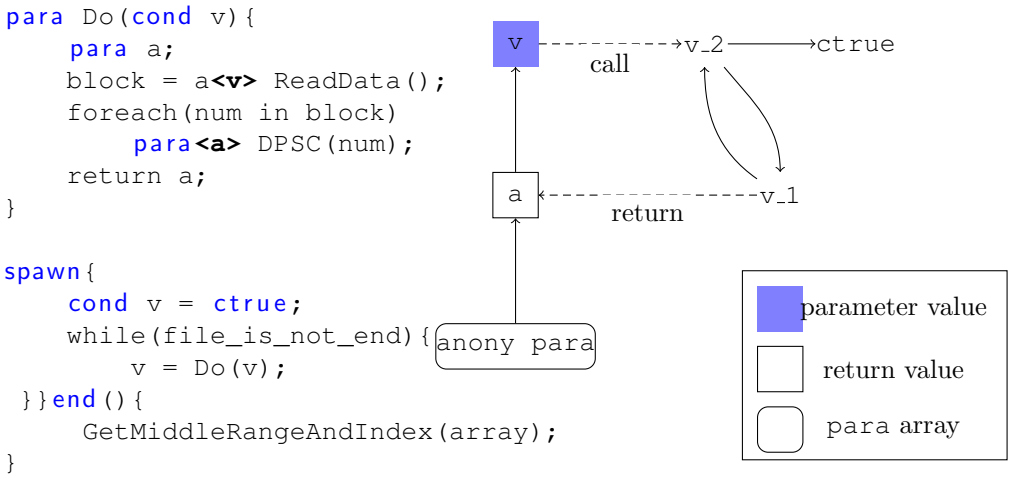

Figure 2: A Static Dependency Graph of `Function Flow`

- `if` chunk has not been encountered before `then` compress it with Ziv-Lempel `else` retrieve its compressed value from the hash
- write the compressed chunk out to the output stream.

This is a typical example of a pipeline that must be set up dynamically because a selection between tasks is done at run-time. The traditional approach is to use a blocking queue to save the result of each stage but the problem with this is that two successive stages cannot be combined into one if the second depends on a run-time selection. Instead, the branches of the selection must be merge into one, giving a coarse-grained parallelism and possibly a severely inefficient load imbalance.

Implemented in functional wait, however, the code is as follows.

```
vector<para> c(frag_counts);
for(int i=0; i < frag_counts; i++)
{
  para a, b;
  Hash hash = a GetHashValue(frags[i]);
  if(!IsExisted(hash))
  {
    b<a> FindDataViaHash(hash);
  }
  else
  {
    b<a> CompressData(frags[i]);
  }
  if(i == 0)
    c[0]<b> WriteBack(cmpddata[i]);
  else
    c[i]<b && c[i-1]>
            WriteBack(cmpddata[i]);
```

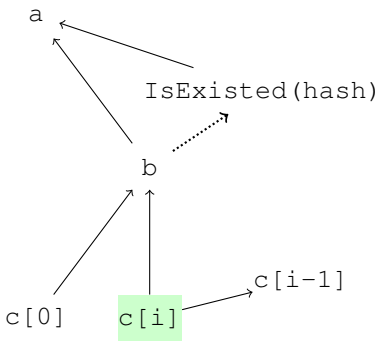

Figure 3: SDG of `Dedup`.

```
}
```

Even the code is fine grained, Optimising opportunities are still reported by our analyzer. The SDG is shown in Fig. 3. Our analyzer reports that `IsExisted` may cause decreased parallelism and possible task fusion. Although `Function Flow` promise that `IsExisted(hash)` can be evaluated correctly, the evaluation is necessary because it is part of control flow. This makes the code before and after `IsExisted` sequentially executed. Noticing that `a`, `b` and `IsExisted` are tightly coupled, we can apply task fusion here to combine them into one task which rebuild parallelism. Here is our optimised code.

```
void loop_wrapper(int index)
{
  Hash hash = GetHashValue(index);
  if(!IsExisted(hash))
  {
    FindDataViaHash(hash);
  }
  else{
    CompressData(frags[i]);
  }
}
vector<para> c(frag_counts);
for(int i = 0; i < frag_counts; i++)
{
  para s;
  s loop_wrapper(i);
  if(i == 0)
   c[0]<s> WriteBack(cmpddata[i]);
 else
   c[i]<s && c[i-1]>
           WriteBack(cmpddata[i]);
}
```

Dynamic pipeline is common in many fields, such as X264 in video processing, data mining, etc. In this kind of pipeline, dependences between stages is decided by runtime values and this makes dynamic pipeline cannot be pro-

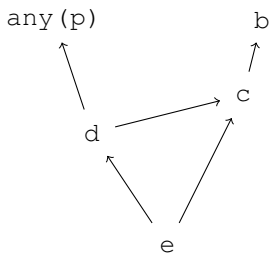

Figure 4: SDG of C Lexical analyzer.

grammed statically. So bug detection and optimising is more difficult than static pipeline.

4.2 C Lexical Analysis

Lexical analysis is the initial phase of compilation in which a stream of characters is converted into a stream of tokens. As the characters are read in one by one, a finite state machine (FSM) is traversed. An FSM is a directed graph of states as nodes and transitions between states as edges. Each transition x →y is labelled with the character that causes the state to change from x to y. The traversal starts at an initial state and ends in one of many final states, each of which emits a token, whereupon it resets to the initial state. Although the current state is always needed to calculate the next state, an unavoidable data dependence, we can use speculative parallelism to calculate the next-but-one state, by taking possible successive states as inputs. However, this approach causes uneven loads, so we await only the fastest state. The following code implements this idea and cooresponding SDG is shown in Fig. 4.

```
vector<String> vs = inferrStates(prefix);
vector<para> p(vs.size());
Ast *fast = NULL, *tast, *rast;
for(int i=0; i < vs.size(); i++)
{
  p[i] buildSubAst(vs[i], fast);
}
para b, c, d;
String token= b parseNextToken(prefix);
c<b> buildSubAst(token, tast);
d<any(p) || c>
    checkSemantics(fast, tast, rast);

bool is_spec_right =
   e<c && d> verifySpec(rast, tast);
if(!is_spec_right)
   checkSemantics(NULL, tast, rast);
```

Our analyser reports opportunities for code motion and task fusion. As task c's termination will trigger more task's, emitting c firstly is better. So we can foward code of creating task b and c. Besides, b is only dependent by c, task

b and c can be fused as one task. The following is our optimised code.

```
void fusion_b_c(String pre, Ast *t)
{
  String token= parseNextToken(pre);
  buildSubAst(token, t);
}
...
para w;
Ast *fast = NULL, *tast, *rast;
w fusion_b_c(prefix, tast);

vector<String> vs = inferrStates(prefix);
vector<para> p(vs.size());

for(int i=0; i < vs.size(); i++)
{
  p[i] buildSubAst(vs[i], fast);
}
para d;
d<any(p) || w>
    checkSemantics(fast, tast, rast);

bool is_spec_right =
   e<w && d> verifySpec(rast, tast);
if(!is_spec_right)
   checkSemantics(NULL, tast, rast);
```

Typically, this kind of applications can be abstracted as a problem in which transitting from current state to the next states needs heavy computation cost. Speculative parallelization is used to bypass the heavy computation and take possible successive states as inputs. Unfortunally, taking various successive states as inputs cause various load. Thus, we take several successive states as inputs parallel and only wait the fastest one.

5 More details

As `Function Flow` is still under heavy development, the best way to get details of `Function Flow` is sending mails to us. But you may want to read the following documents before that.

1. `Function Flow` language reference.
2. Source to source compiler.
3. Static Analyzer.
4. Optimizer.
5. Runtime.

6 Publications

Function Flow: Enforcing Compiler for Task Parallelism via Language Builtin Support. (submit to OOPSLA)

Xuepeng Fan, Hai Jin, Liang Zhu, Xiaofei Liao, Chencheng Ye, Xuping Tu: Function flow: making synchronization easier in task parallelism. PMAM 2012: 74-82

7 Download

source code here!