Embla: a Tool for Dependency Profiling

Karl-Filip Faxén   Lars Albertsson

August 28, 2006

Abstract

With the widespread use of chip multithreaded (CMT) processors, there is an urgent need for tools and methodologies supporting parallelization of existing applications. Writing explicitly threaded code is a challenging task, especially since such code is naturally nondeterministic.

In this paper we present a novel tool supporting the process of parallelizing programs by hand. The tool, Embla, allows the user to profile the dependencies in a sequential program, thus finding opportunities for parallelization. While most tools for this problem rely on static analysis, Embla is a profiler that records dependencies as they arise during program execution. It is thus exact in contrast to static tool which by necessity make conservative approximation. Also, since the tool deals with the machine code of the program, it is completely language independent.

1   Introduction

Ubiquity of chip multithreading

The need for thread parallelism

A parallelisation model: asynchronous function calls

Embla is, at least initially, aimed at supporting procedure-level fork-join parallelism. To a first approximation, this means executing procedure calls asynchronously. Consider the program fragment below:

```
foo();
bar();
baz();
```

Suppose that `foo()` and `bar()` are *independent*, ie that executing `bar()` before `foo()` does not change the meaning of the program[1]. A sufficient condition for independence in this case is that `foo()` and `bar()` do not do any I/O and that neither call writes to a memory location that the other call accesses. In that case the call to `foo()` can be asynchronously, by a different thread, and in parallel with the call to `bar()`. Suppose further that `foo()` and `baz()` are not

---

[1]There are many ways to formally give meaning to a program, although it has never been done for C.

independent, so `baz()` can not be executed until the call `foo()` is completed. We can express this as

```
spawn foo();
bar();
sync;
baz();
```

where `spawn` starts the call in parallel and `sync` waits for all `spawn`'d acivities to terminate (here we have borrowed programming constructs from the Cilk programming language [1]).

This style of paralleism can be implemented efficiently (as is done for instance in the Cilk language) and is easy to understand although not as powerful as general thread parallelism with arbitrary synchronization. In particular, as long as `foo()` and `bar()` are independent, the behavior of the parallel program is identical to the behavior of the sequential version. Therefore it is sufficient to understand (debug, verify, . . .) the sequential program; everything except performance carries over to the parallel version.

The price for this approach is that one must find independent procedure calls (program fragments, in general). Thus there must be such calls in the code and the programmer must be able to make sure that they are in fact independent. The first issue depends of course on the algorithms used in the program.

The second issue is typically dealt with using static analysis tools as is done in parallelizing compilers. Such analysers are however very complex and must by necessity be conservative (since they are static, they must use approximations which are always safe).

Embla takes a different approach by observing program execution directly and recording the dependencies that occur.

2   Overview of the Implementation of Embla

In order to understand dependency profiling with Embla, we here give an overview of the implementation of Embla.

Embla is based on instrumented execution of binary code. Although our examples of profiling output use a high level language (C), the profiling itself is on instruction level, followed by mapping the information to source level using debugging information in the standard way.

Embla uses the Valgrind instrumentation infrastructure, so there is no offline code rewriting; the Embla tool behaves like an emulator of the hardware.

There are three fundamental data structures in Embla.

The memory table: Contains information about the most recent reads and writes to each block of memory. Blocksize is a tradeoff between memory use and precision.

The activation tree: Contains a record for each stack frame allocated during execution. The rightmost branch of the activation tree corresponds to the stack.

The result table: A hash table mapping dependency signatures to counters of RAW, WAR and WAR dependencies. Dependency signatures identify the source and target of the dependency as well as other information of interest.

3 The Tool

Embla helps the user find dependencies in a program. Technically, a dependece is two references, not both reads, to overlapping memory locations with no interveaning write. What makes Embla special is the way dependencies are reported. Since the idea is to support manual, function call level parallelization, we are not really interested in knowing that a memory read on line 11754 in function `foo` sometimes reads a value last written in line 3411 in function `bar`. Rather, the user is concerned with a completely different function `baz` and would like to know whether the call to `foobar` on line 2355 can be made in parallel with the call to `barfoo` on line 2356.

3.1 Types of dependencies

There are several different types of dependences between two references. These types can be characterized according to several dimensions. The first dimension concerns the read and write nature of the references:

Flow: A true data dependency wher the location is first written then read. Also known as *read after write* (RAW).

Anti: A dependence caused by reuse of a location that is first read and then written. Also known as *write after read* (WAR).

Output: Similar to an anti dependence, but the second reference is also a write. Also known as *write after write* (WAW).

Some anti and output dependencies are due to reusing the same locations for (logically) distinct data items. This happens for instance for stack locations when arguments are passed on the stack (as is common on the x86, for instance). Consider the following example:

```
foo(x);
bar(y);
```

The value of `x` will be pushed on the stack. Then `foo` will be called and will read `x`. When that call has returned, `y` will be pushed on the stack on the same address as `x`, creating an anti dependence with `foo`'s read. This kind of dependence can easily be removed by a compiler. In fact, if the call to `foo` is made

asynchronously, in another thread, it will automatically be made on a different stack and the dependence disappears. Register allocation is another source of anti and output dependencies since distinct variables are packed into the same register. In this case, the allocation can be changed to remove dependenies, a transformation known as *renaming*. Other anti dependencies are due to updates to the same data structure and cannot be eliminated by renaming.

Another dimension that Embla recognizes is where in memory the location associated with the dependency is situated. In particular, we want to catch the cases where anti and output dependencies are created by reuse of memory for new stack frames. This happens when the stack shrinks and then grows again; the new stack frames use the same memory locations, but only for memory management reasons. With this in mind we have the following categories:

s(tack): The location has been a part of the stack from the first reference to the second (inclusive).

f(alse): The location was part of the stack when the first reference was made, then the stack shrunk enough that the location was not part of the stack and then it became part of the stack again.

o(ther): The location has never been part of the stack.

We also categorize dependencies according to properties of the two memory references making up the source and target of the dependence. This is both in order to give more precise information and to flag additional dependencies as spurious.

A typical case in point is `malloc`. Each call to `malloc` manipulates (updates) administrative data structures like the free list. Embla will report these as dependencies, effectively serializing all calls to `malloc`, and thus in addition all calls to functions calling `malloc`. In reality, these calls need not be serialized since program semantics typically does not depend on the exact addresses that data are allocated. It suffices that a thread safe implementation of `malloc` is used.

Dependency endpoints correspond to instruction execution events and are reported as line numbers with one of the following codes:

ϵ: No code at all represents an instruction that is generated from the source code at the indicated line. Thus the instruction is part of the function or procedure containing the line.

c: The instruction was executed by a function (procedure) that was (transitively) invoked by a function (procedure) call at the indicated line.

h: Like **c**, but the function containing the instruction was *hidden*, that is, it was on a blacklist of functions which, like `malloc`, would not give rise to dependencies in the parallelized program.

4

```
#include <stdlib.h>
#include <stdio.h>

struct {int a; int b;} has_ab;

int main(int argc, char **argv)
{
    has_ab.a = 1;
    has_ab.b = 1;
    printf("%d ", has_ab.a);
    has_ab.a = 2;
    has_ab.b = 2;
    printf("%d %d\n", has_ab.a, has_ab.b);

}
```

References

[1] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

```c
#include <stdio.h>
#include <stdlib.h>

#define N 5

typedef struct _ilist {
    int val;
    struct _ilist *next;
} ilist;

static ilist* mklist(int n)
{
  int i;
  ilist *p = NULL;

  for( i=0; i<n; i++ ) {
    ilist *t = (ilist *) malloc( sizeof(ilist) );
    t->val = i;
    t->next = p;
    p = t;
  }
  return p;
}

static int sumlist(ilist* p)
{
    ilist *q;
    int  sum = 0;

    for( q=p; q!=NULL; q=q->next ) {
        sum += q->val;
    }
    return sum;
}

int main(int argc, char **argv)
{
  int m,n;
  ilist *p,*q;

  p = mklist( N );
  q = mklist( N );
  m = sumlist( p );
  n = sumlist( q );

  printf("%d\n", m+n);

  return 0;
}
```
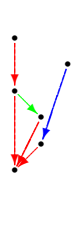
```c
#include <stdlib.h>
#include <stdio.h>

struct {int a; int b;} has_ab;

int main(int argc, char **argv)
{
    has_ab.a = 1;
    has_ab.b = 1;
    printf("%d ", has_ab.a);
    has_ab.a = 2;
    has_ab.b = 2;
    printf("%d %d\n", has_ab.a, has_ab.b);

}
```

```c
#include <stdio.h>
#include <stdlib.h>

#define N 5

typedef struct _ilist {
    int val;
    struct _ilist *next;
} ilist;

static ilist* mklist(int n)
{
  int i;
  ilist *p = NULL;

  for( i=0; i<n; i++ ) {
    ilist *t = (ilist *) malloc( sizeof(ilist) );
    t->val = i;
    t->next = p;
    p = t;
  }
  return p;
}

static int sumlist(ilist* p)
{
    ilist *q;
    int  sum = 0;

    for( q=p; q!=NULL; q=q->next ) {
        sum += q->val;
    }
    return sum;
}


int main(int argc, char **argv)
{
  int m,n;
  ilist *p,*q;

  p = mklist( N );
  q = mklist( N );
  m = sumlist( p );
  n = sumlist( q );

  printf("%d\n", m+n);

  return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>

static int nfib(int n)
{
    int result;
    if( n < 2 ) {
        result = 1;
    } else {
        int a = nfib( n-1 );
        int b = nfib( n-2 );
        result = a+b;
    }
    return result;
}


int main(int argc, char **argv)
{
    int m = nfib( 8 );

    printf( "%d\n", m );
}
```

```
#include <stdlib.h>
#include <stdio.h>

static int nfib(int n)
{
    int result;
    if( n < 2 ) {
        result = 1;
    } else {
        int a = nfib( n-1 );
        int b = nfib( n-2 );
        result = a+b;
    }
    return result;
}


int main(int argc, char **argv)
{
    int m = nfib( 8 );

    printf( "%d\n", m );
}
```

```c
#include <stdlib.h>
#include <stdio.h>

static int a[]
  = {17, 3, 84, 89, 4, 5, 23, 43,
     21, 7, 2, 1, 55, 63, 21};
static int n = 15;

static int *part(int *a, int n)
{
    int i = a[0];
    int k = a[n-1];
    int *lp = a;
    int *hp = a+n-1;

    while( lp<hp ) {
        if( k<i ) {
            *lp = k;
            lp++;
            k = *lp;
        } else {
            *hp = k;
            hp--;
            k = *hp;
        }
    }
    *lp = i;
    return lp;
}

static void qs(int *a, int n)
{
    if( n>1 ) {
        int *lp = part( a, n );
        int m = lp-a;
        qs( a, m );
        qs( lp+1, n-m-1 );
    }
}

int main(int argc, char **argv)
{
    int i;

    qs( a, n );
    for( i=0; i<n; i++ ) {
        printf( "%d ", a[i] );
    }
    printf( "\n" );

}
```

```c
#include <stdlib.h>
#include <stdio.h>

static int a[]
  = {17, 3, 84, 89, 4, 5, 23, 43,
     21, 7, 2, 1, 55, 63, 21};
static int n = 15;

static int *part(int *a, int n)
{
    int i = a[0];
    int k = a[n-1];
    int *lp = a;
    int *hp = a+n-1;

    while( lp<hp ) {
        if( k<i ) {
            *lp = k;
            lp++;
            k = *lp;
        } else {
            *hp = k;
            hp--;
            k = *hp;
        }
    }
    *lp = i;
    return lp;
}

static void qs(int *a, int n)
{
    if( n>1 ) {
        int *lp = part( a, n );
        int m = lp-a;
        qs( a, m );
        qs( lp+1, n-m-1 );
    }
}


int main(int argc, char **argv)
{
    int i;

    qs( a, n );
    for( i=0; i<n; i++ ) {
        printf( "%d ", a[i] );
    }
    printf( "\n" );

}
```