

Parallelization of Legacy Telecom Software

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

`bjorn.lisper@mdh.se`
`http://www.idt.mdh.se/~blr/`

2008-09-12

Multicore – The Software Problem

Efficient use of Multicore requires parallel execution

So far, (mainstream) processors have been sequential

Thus, current software is constructed to run efficiently on sequential processors

We're facing an enormous problem: how to get legacy code to work well on Multicore processors

I will talk about this problem in general

Then zoom in on telecom software

Experiences from a joint research project with Ericsson

Parallelization

Parallelization – transform programs automatically into running in parallel

Success critically dependent on:

- inherent parallelism in the application
- inherent parallelism in the selected solution (algorithm, data representations, ...)
- absence of features in the programming language that obscure parallelism

Has been attempted in the supercomputing domain for many years – only partially successful

I don't believe it will work in general, only in certain niches

What's the Problem?

Most programming languages (C, C++, java, ...) have a *memory concept* (simple program variables, objects, arrays, etc).

Computed values are *stored* in memory and *read back* when used

This gives *dependences* in the code:

x = ...
...

y = ...x...

Read-after-write
(RAW)

x = ...y...
...

y =

Write-after-read
(WAR)

x =
...

x =

Write-after-write
(WAW)

Statements must be executed in order \implies **parallel execution might go wrong!**

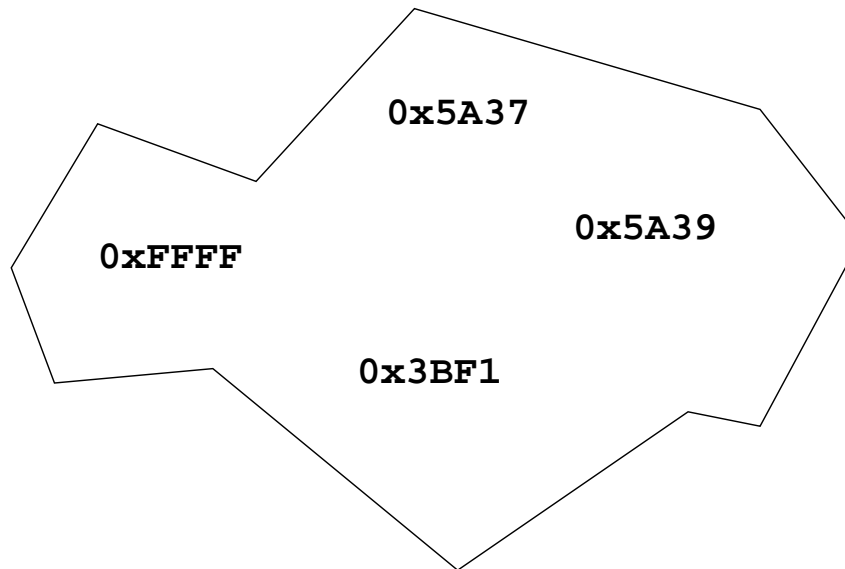
Only statements **surely** without dependences can be run in parallel

It's Even Worse

Consider this C assignment:

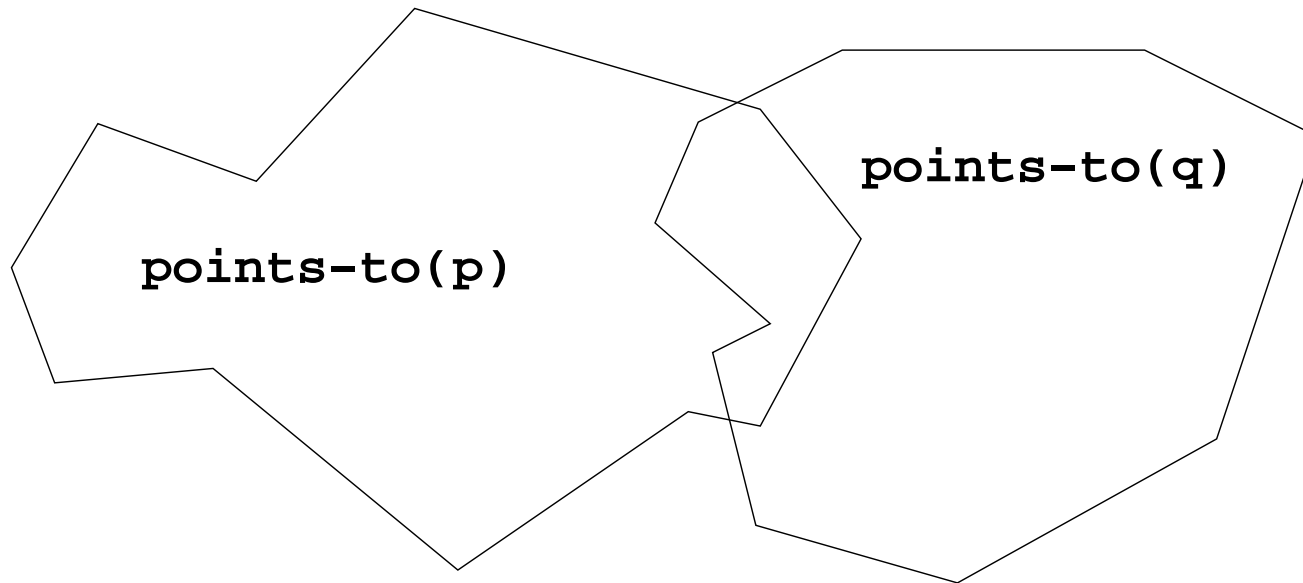
```
*p = .....
```

`p` is a pointer: it might point to many different addresses:



It's Even Worse, Part II

Possible dependency between $*p$ and $*q$ if their *points-to* sets intersect:

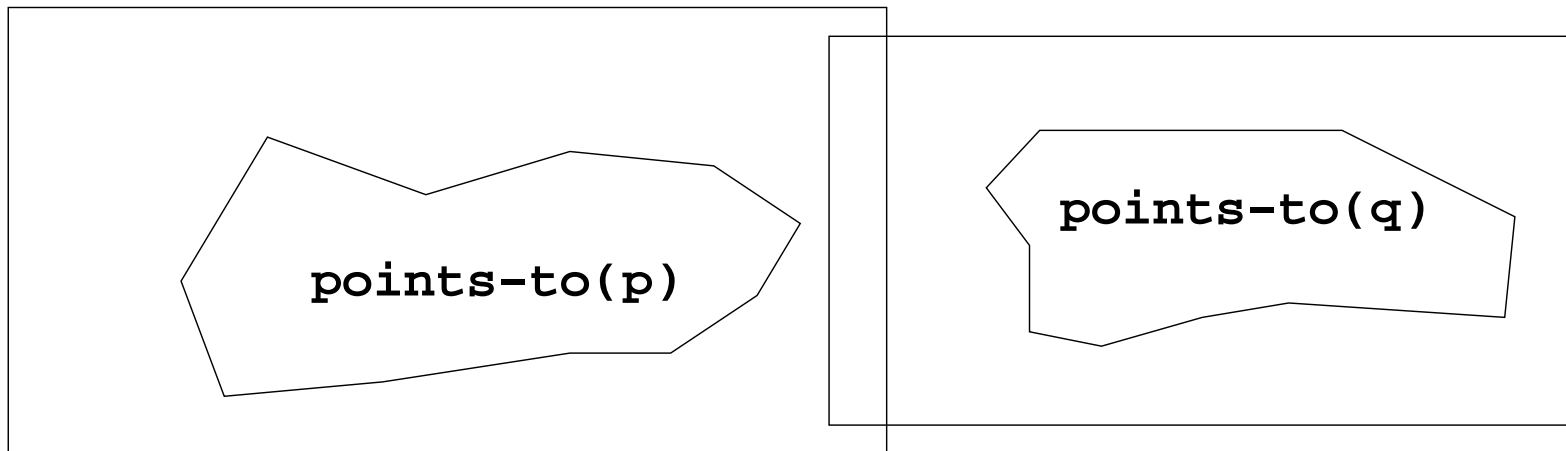


It's Even Worse, Part III

points-to sets can be found automatically

The analysis must find **all** addresses in the set to be **safe**

Such an analysis might have to *overestimate* the set: can give detection of false dependences



It's Even Worse, Part IV

So much for C

What about Object-Orientation?

The good:

Objects localize state: accesses in different objects cannot yield dependences

The bad:

Objects can be created and accessed in a very *dynamic* fashion, may be very hard to predict which object is being accessed

Subclassing & virtual methods pose a problem. May be unknown until runtime which class an object belongs to, or which method is actually being called. Very hard then to find the possible memory accesses, and thus dependences, in advance

The ugly:

C++: combined difficulties of high-level OO features and low-level C stuff
(pointer arithmetics, no type safety, ...)

Conclusion, General Case

Automatic parallelization of legacy code won't work in general

(No silver bullet: you'll rather have to bite the bullet)

However, there might be niches where special techniques can work well

A possible such niche: telecom exchange software

Telecom Systems

A telecom exchange is really some kind of server

Lots of incoming requests (dialup, close a call, perform some service, ...)

These request are *highly concurrent!*

As long as two requested services *do not touch the same parts of memory* they can be performed in parallel!

Parallelization of telecom software means to retrieve this concurrency, and exploit it

Dependences vs. Conflicts

An important difference to conventional parallelization:

“Classical” parallelization concerns fully sequential code in a single thread

Possible dependence between statements means they must be executed in original (sequential) order

Parallelization of concurrent activities means to analyze for freedom of possible *conflicts*

No conflict \implies can be executed fully in parallel

Possible conflict \implies can be executed in any order, but not concurrently

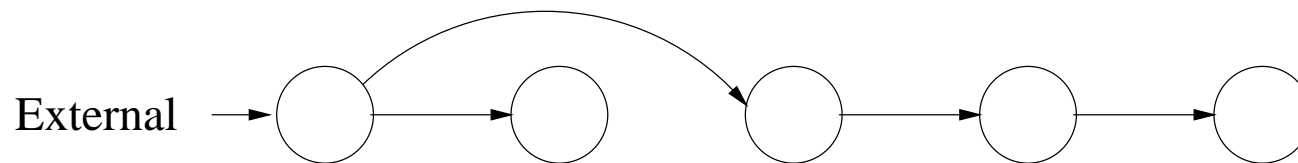
Event-driven Software

Telecom software is often *event-driven*

An event triggers a *job* – a terminating execution of some code

Events can be external, or generated by jobs

A “job-tree”: the list of jobs originating from some external event



Jobs in *different job trees* are typically concurrent, and can be executed in parallel if no conflicts!

Embedded Systems Software

A side note:

Embedded systems software often have a similar structure

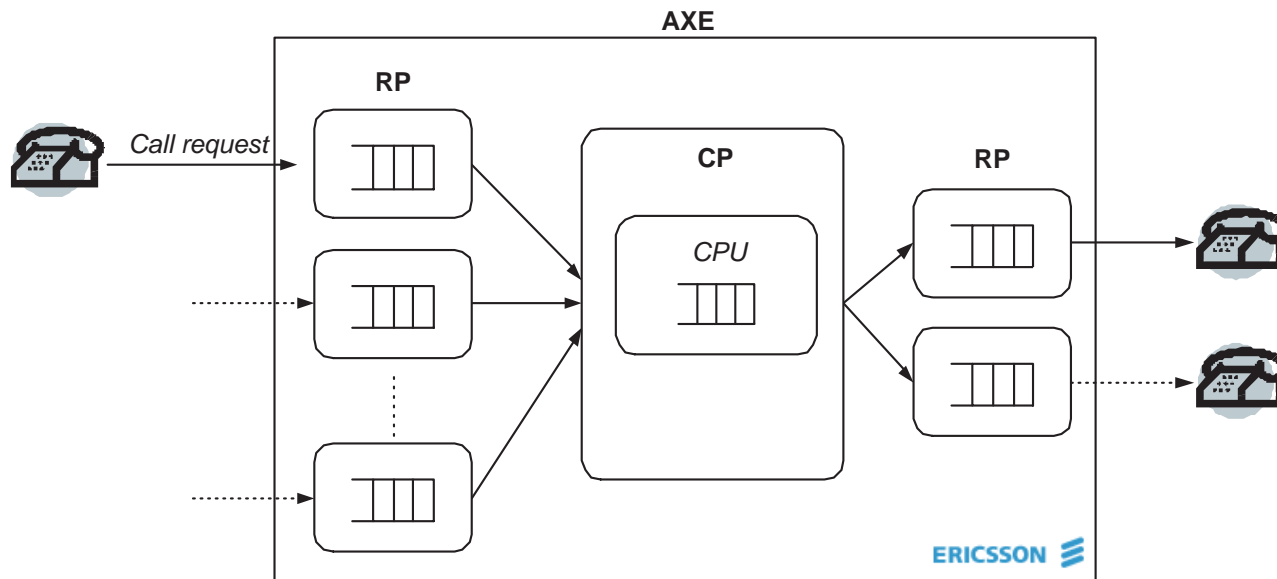
“Tasks” (in real-time systems terminology) are same as jobs

Tasks may be time-triggered, but occurrence of time can be seen as an event

Seems like parallelization by conflict analysis should be of interest also for embedded systems

A Case: AXE/PLEX

AXE: Ericsson's classical telephone exchange



The project concerned parallelization of the SW for the CP

AXE Software

AXE is programmed in PLEX, and has a particular run-time system

Jobs are triggered by *signals* (external or generated by other jobs)

Jobs (user-level) are put in a simple FIFO queue in the order they are triggered

Jobs in the queue are executed in that order

A job is executed to completion before the next job starts (no preemption)

Some Consequences of the Run-Time System Model

User-level code will never be interrupted by other user-level code

Programmers might have relied on this

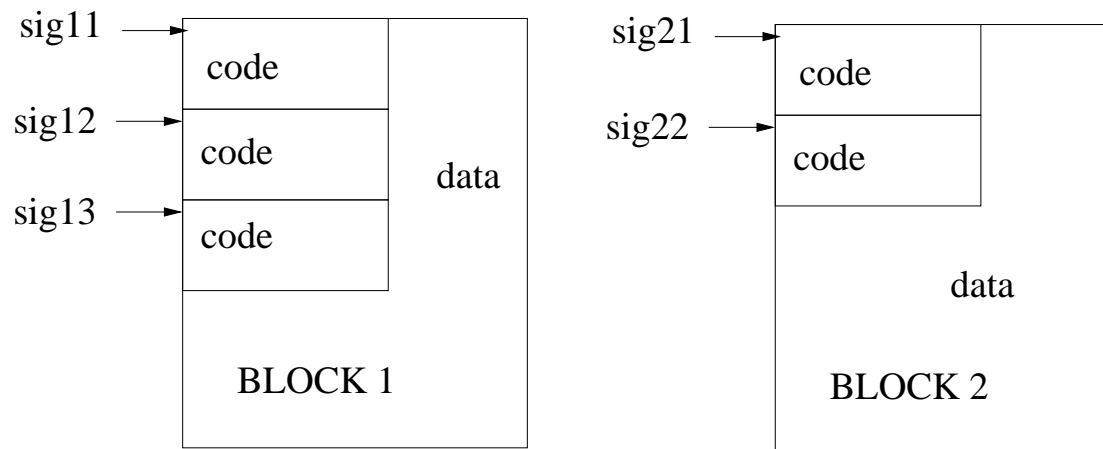
Common data might therefore be unprotected

If jobs execute in parallel and access the same parts of memory, they may thrash each other's data!

Parallelization of PLEX code thus requires a conflict analysis, to see if jobs can possibly access the same memory areas

PLEX' Memory Model

PLEX code is structured into *blocks* containing code and data:



Data can only be accessed by code in the same block

Code in different blocks can thus always execute in parallel!

Relation between Blocks and Objects

Blocks can be seen as objects

Signals can be seen as methods

However, blocks are *statically allocated*

No class system, thus *no subclassing* and *no virtual methods*

All these properties *simplify parallelization*

Blocks can also be seen as *components* carrying code and data

A First Approach to Conflict Handling

One lock per block

A job entering a block acquires the lock, and releases it at exit

Simple, safe! Jobs in different job trees can then be run in parallel

Has already been tried by Ericsson

However, turns out that some blocks become “hot spots”

This resulted in insufficient speedups!

A Simple Conflict Analysis

Blocks can be more closely analyzed w.r.t. memory conflicts

For each signal, its code can be scanned for:

- Variables that are possibly read
- Variables that are possibly written

Possible conflict between two signals if both can access the same variable, and one can possibly write it

Yields a *conflict matrix* telling which signals are in possible conflict

Jobs calling signals not in conflict can execute in parallel, even if in the same block!

Analyzability of PLEX

PLEX is quite amenable to this kind of “memory access analysis”:

- Lacks the difficult OO features (class system, dynamic object allocation)
- No pointers!
- No dynamic memory handling (more or less)
- No dynamic jumps (computed GOTOs, function pointers, ...)

A Code Study

We analyzed four representative blocks by hand to find their conflict matrices:

- Fraction of (possibly) conflicting signals in range 33% – 91 %
- With *variable privatization* code transformation: 8% – 65 %
- Also with *protected SEIZE* transformation: 0% – 65 %

Indicates that a simple static analysis like this actually can be helpful

However, more research needed (also the *time* spent executing different signals affects speedup)

Alas, the project was terminated at this point ☹️

Conclusions

Automatic parallelization of legacy code is a pipe dream in general

May work for special applications, which have enough inherent concurrency

Telecom is such an application

We investigated a simple static conflict analysis for PLEX software on the AXE system

Promising results, more research needed

For best results, I believe the static analysis should be combined with some dynamic conflict resolution technique that handles the remaining conflicts