CoRReT: A Constraint Based Environment for Rapid Prototyping Real Time Programs

*

Krishna V. Palem
Courant Institute of Mathematical Sciences, NYU

Abstract

The information revolution that we are in the midst of has led to the use of computers controlling applications ranging from automobiles and games, to video-pumps in the information highway. These applications are distinguished by the fact that they use programs with special timing relationships between their constituent elements. For example, a program running in the microprocessor controlling an *ABS* system in a modern automobile must sense and react to the friction coefficient between the brake pads and the wheel at well-defined intervals of time; failure to do so will result in a systemic failure of the brakes. Referred to typically as *embedded systems*, these applications constitute a significant portion of the potential growth in the computer industry. However, this growth opportunity is being hampered by a lack of adequate support via software development tools, to aid the *easy, rapid* and *correct* prototyping of embedded applications.

In this report, we outline *CoRReT*, a *co*nstraint based environment for the *r*apid prototyping of *re*al *t*ime programs. The report outlines the overall system architecture as well as the key modules in this environment that are being currently developed. CoRReT is

*Supported in part by an award from Hewlett-Packard Corporation, from IBM corporation, and a NYU research challenge grant.

a scheduling centric system in that a suite of algorithms for *instruction scheduling* programs instrumented with real-time constraints, are at its core . These algorithms are an integral part of an (optimizing) compiler which will compile these programs *automatically* while attempting to ensure that the timing constraints are met; when the constraints are met, the resulting *schedule* for the instructions is referred to be *feasible*. If a feasible schedule is found, it will be fed automatically into a code-generator in the back-end of the compiler. Our envisioned scheduler can — in addition to traditional control- and data-dependence constraints in the source program — also cope with a variety of timing constraints specified by the programmer.

Our focus is on computational platforms that embody parallelism at two levels of granularity. At the highest level, we envision a tightly-coupled parallel machine offering large-scale parallelism. In this setting, a single embedded application can be distributed across the individual processors of the cluster. Furthermore, each processor in this parallel machine can embody *I*nstruction *L*evel *P*arallelism (*ILP*) at a fine-grained level.

Unfortunately, due to a lack of automatic tools and technology that can provide compilation support for real-time constraints ubiquitous to embedded applications, parallel computing platforms have not proliferated in this setting. Considering the fine-grained case first, RISC processors with ILP have not yet found a niche in this domain; currently, developers of embedded systems are reluctant to embrace ILP technologies due to the onerous task of ensuring timing relationships in the program by hand — a difficulty compounded by parallelism (at a fine-grained level) in the processor. Clearly, providing support through automation that frees the programmer of these difficulties, is a means of overcoming this challenge.

Our response to this challenge via CoRReT is to develop scheduling methodologies and tools for automatically harnessing very high performance from these platforms, in the context of embedded systems. In the absence of time-constraints, major progress has been achieved in this direction at the coarse-grained level. The situation is even better at the fine-grained level where scheduling technology is being used routinely in product-quality compilers for RISC processors.

The methodology on which CoRReT is based is independent of any particular target processor, and is applicable to third and fourth generation languages. Furthermore, we propose to use the same scheduling engines during the static analysis of the program as well as during compilation. We anticipate this "confluence" in the scheduling algorithms to aid in shorter prototyping cycles, since identical schedules will be used by the analysis tools and the back-end of the compiler to generate code. We envision that the algorithms and tools that go into CoRReT will naturally form an integral part of a full-fledged programming environment for prototyping real-time programs on parallel platforms.

CoRRet: A CONSTRAINT Based Environment for Rapid Prototyping Real Time Programs

Krishna Palem

Courant Institute of Mathematical Sciences

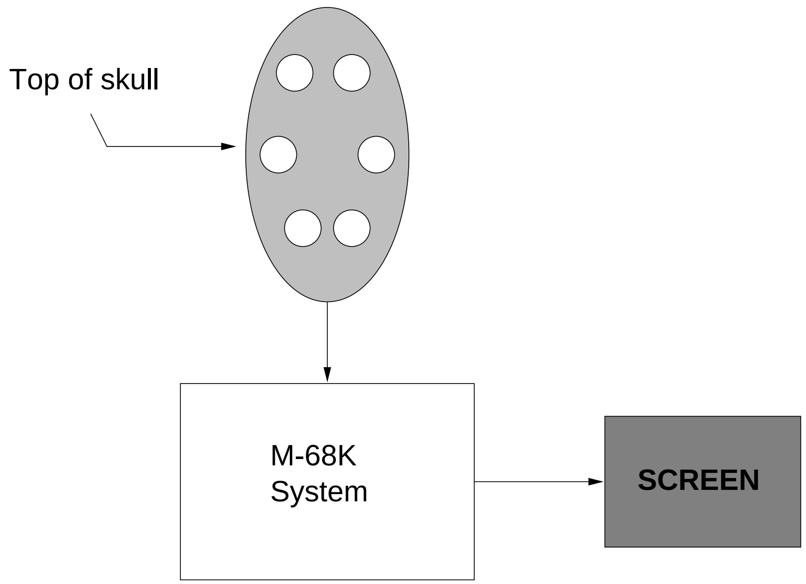
Target Domains for Real-time Applications

1. **Hard Real-Time:** Avionics, medical life-support, ...

2. **Soft Real-Time:** Entertainment (set-top boxes, games), utilities (microwaves, wrist-watches), ...

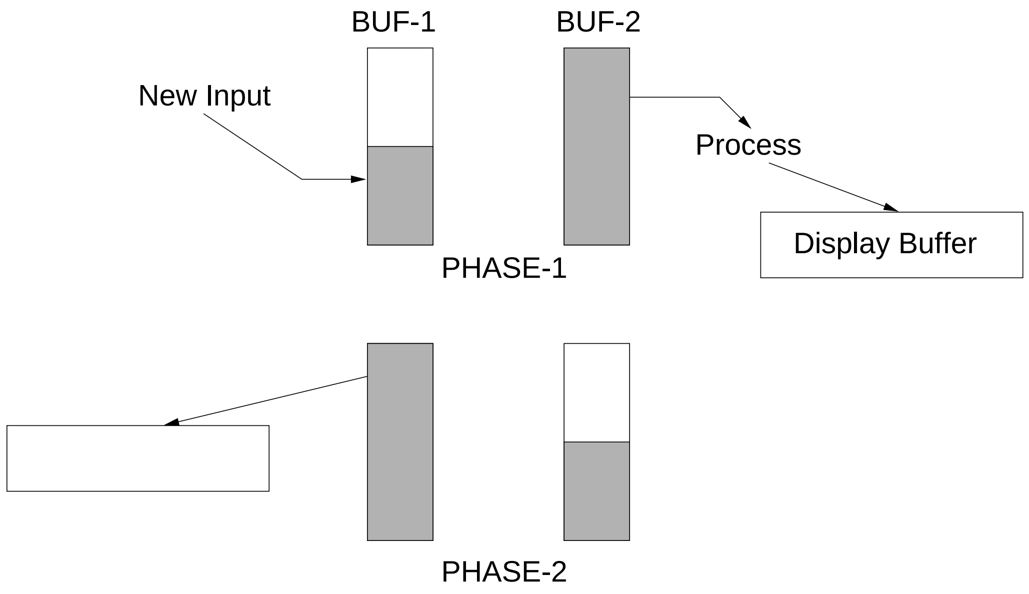
Building These Systems

An example from my experience for real time processing of *Electroencephalographic* data in a doctor's office



- Data is arriving at about 10 KHz.
- Processor clocking about 10 MHz.
- Functions included data acquisition, analysis and real-time display to the neuro-physician.

The High Level Approach



- Collect data in one buffer.
- Process in the other.
- When one of the buffers is full, switch input buffer and processing buffer.

Some Simple Time-constraints

- Processing software has to move ahead by one data point every 0.1 m-sec.

- The switching of buffers has to occur in the same interval.

The Difficulty

- All the coding for the signal-processing had to be done in assembly.

- Had to “hand-tune” it to fit the constraints stated above.

- Other constraints involved interrupts from the board updating the display which was running a dedicated M6800 (8-bit) microprocessor to transfer the processed data over for displaying.

Real-time Software Development in Industry

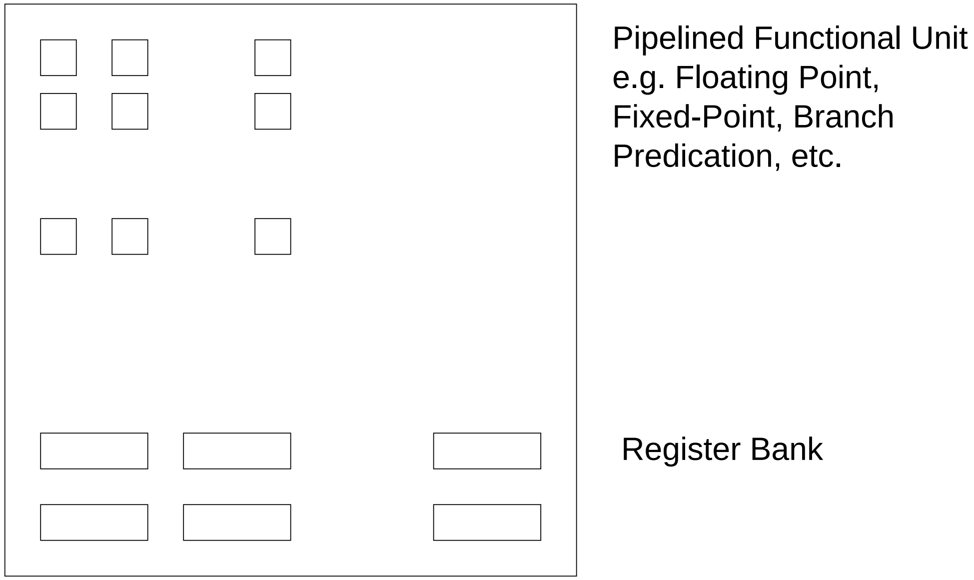
- Programmers spend a lot of time designing pieces of code.
- Estimate timing behavior via
  - synthetic *analysis*, possibly with some tools, and
  - actual measurement where code is produced and executed.
- If timing "expectations" — informally specified — are not met, programs are returned till *feasible*.

Spurred by the RISC Processor Revolution

Opportunity in Superscalars

- High degree of *Instruction Level Parallelism (ILP)* via multiple *Functional Units (FUs)*, each pipelined: Essential to harness promised performance.
- Clean simple model and Instruction Set makes *compile time* optimizations feasible.
- Therefore, performance advantages can be harnessed automatically.

Superscalar (RISC) Processors



Canonical Instruction set

- Load and store to/from memory (multiple cycles).

- Register-Register Instructions (single cycle).
  A few notable exceptions of course.

Eg., IBM Power & RS6K, DEC Alpha, Sun Sparc.

Example Of Instruction Level Parallelism

Processor has

- 5 functional units: 2 fixed point units, 2 floating point units and 1 branch unit.

- Pipeline depth: floating point unit is 3 deep, and the others are 1 deep.

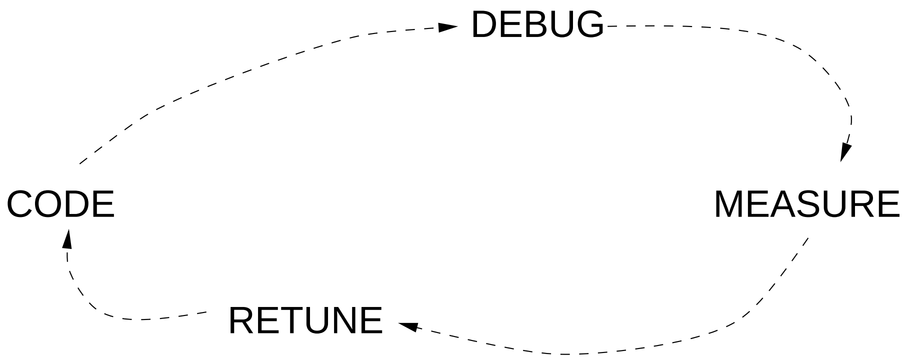
At peak rates, with a 71.5 MHz clock, 357.5 MIPs with 9 instructions being processed simultaneously.

Returning To Real-Time Computing

The technological trend enabling new applications:

- Rapid evolution of processor technology in terms of performance.
- Substantially lower $$ per MIPs.
- Makes much more ambitious real-time applications feasible.
- *Hand-held games* with 32-bit (embedded) microprocessors— *Video-pumps* in the *information highway*.

The Overall Challenge



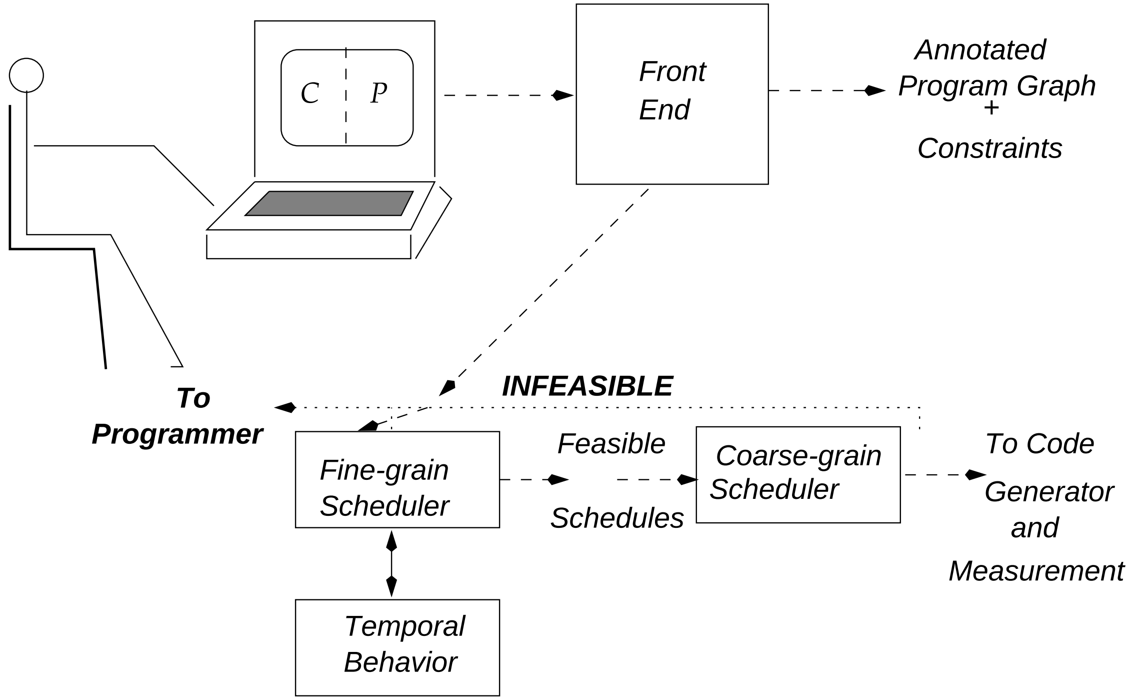
- Error-prone and tedious as the complexity of the system grows.
- Hence substantially superlinear growth in software development cost as the scale grows.
- Limits the scale of the explosion and does not harness hardware potential.
- Further compounded by the need to schedule instructions in the context of modern processors with ILP.

An Important Gap

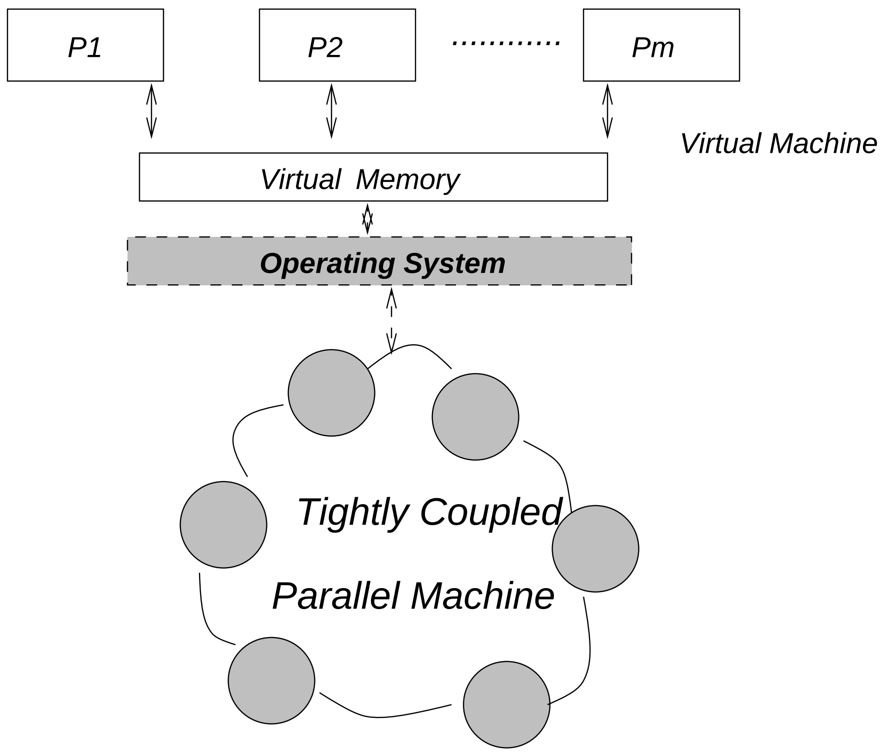
Currently

- Even with some automatic support.
- Methods and algorithms used in analysis are not related to those aimed at compilation.
- Gap between prediction and compiled code's feasibility is large.
- Leads to long prototyping cycles.

The “Big Picture”

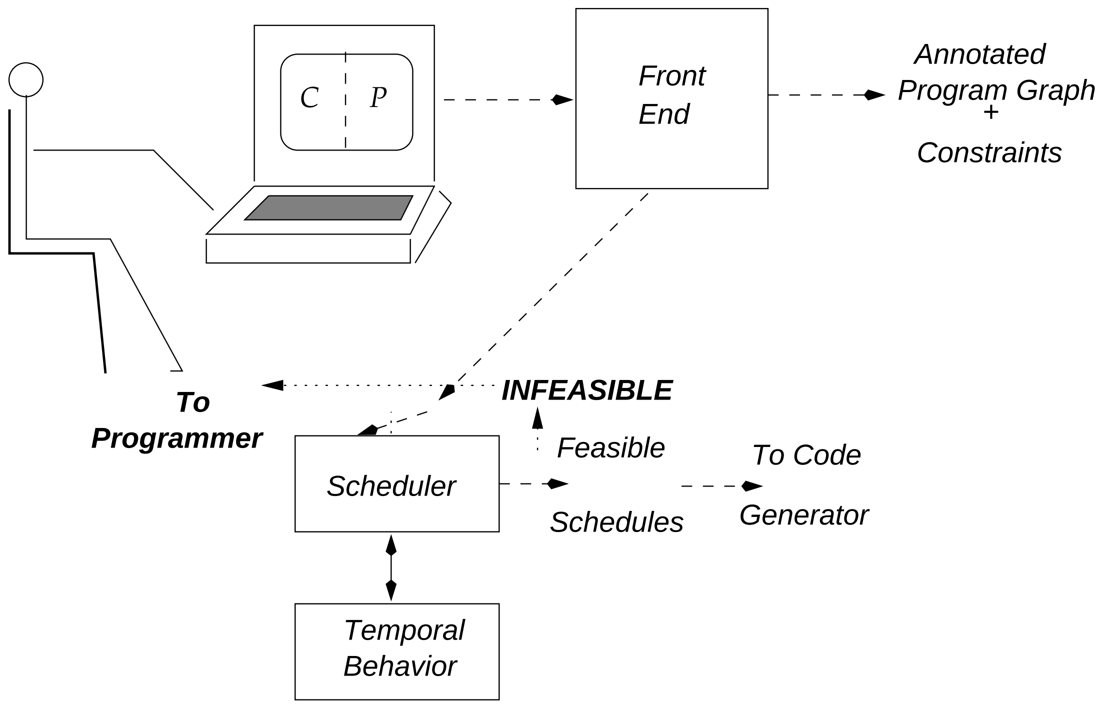


The Platform

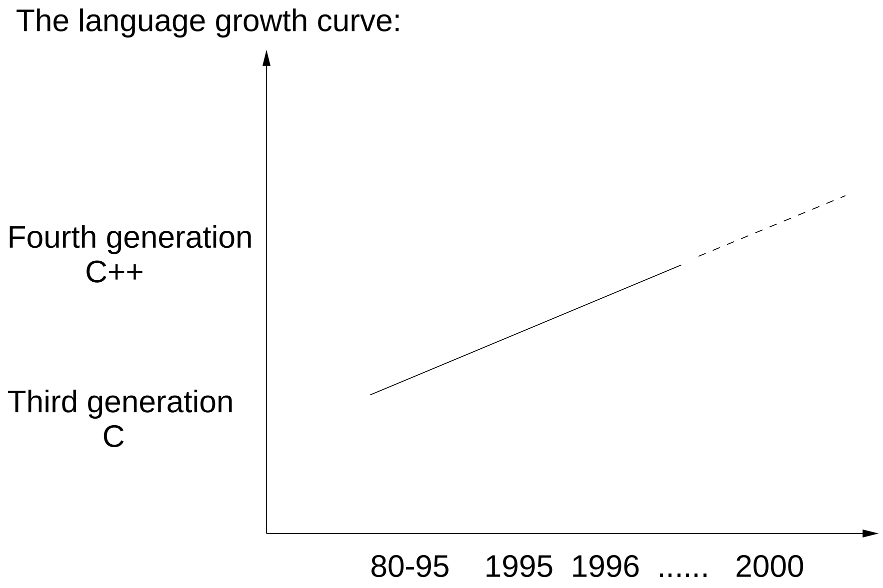


Our Envisioned Response

The Proposed Environment



In Thinking of Solutions



- Programmers are slow to *migrate away* from their favorite and stable languages.

- Radical proposals will not succeed.

- Think of technologies that enhance existing languages in providing automatic support.

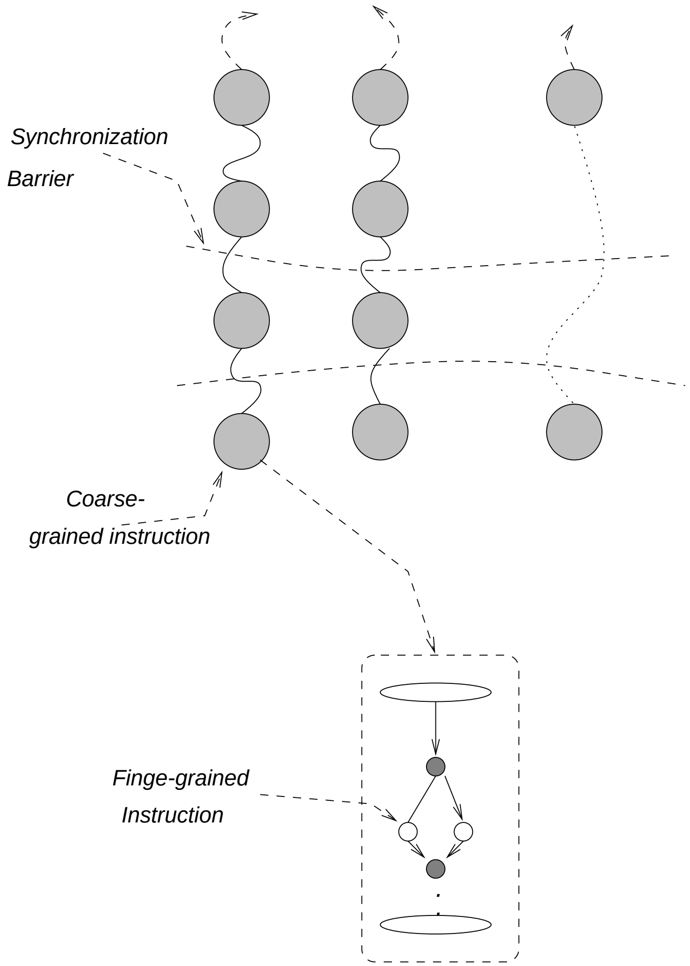
- Consequently, we do *not* propose a new language.

The Technology

Provide Support for

- Easily *expressing* “timing-relationships” between parts of the program.
- Analyze the program to determine feasibility on the target platform.
- Generate code *automatically* if feasible.

The Programming Model

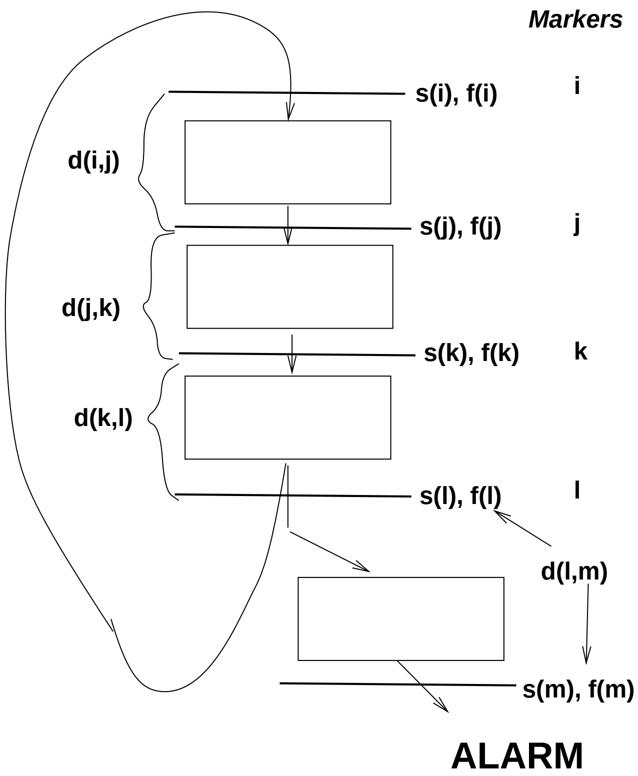


The Proposed Expressive Framework

- Enhance the user program with *zero-time executable markers* shown by thick lines.

- Write timing relationships between markers are a distinct *constraint system*.

The Timing Relationship



Given that **s**, **f** and **d** respectively denote the *start*, *finish* and the *durations* associated with markers as shown, example design specifications — all integers are m-sec — might be

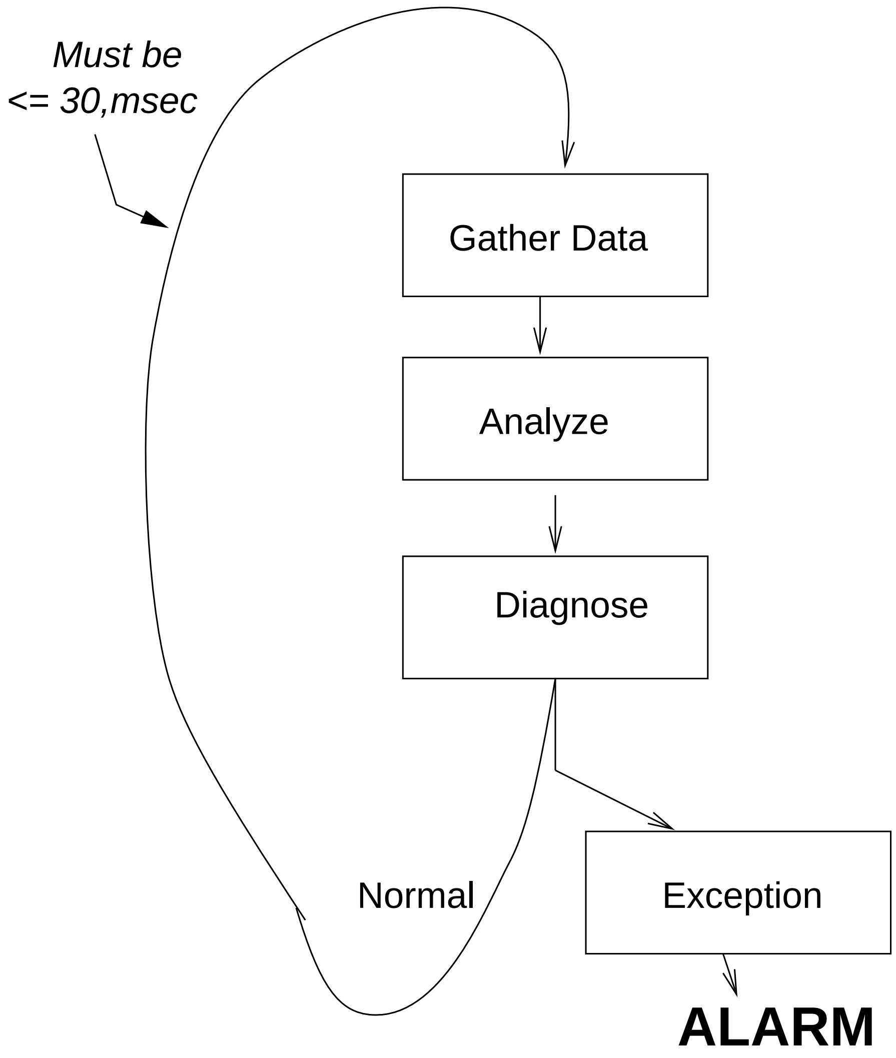
1. $d(i,j) + d(j,k) + d(k,l) \leq 20$

2. $d(l,m) = 50$

3. $f(l) - s(i) \leq 100$

A Simple Example

Cardiac Arrhythmia Detector



Constraints As Time Specifications

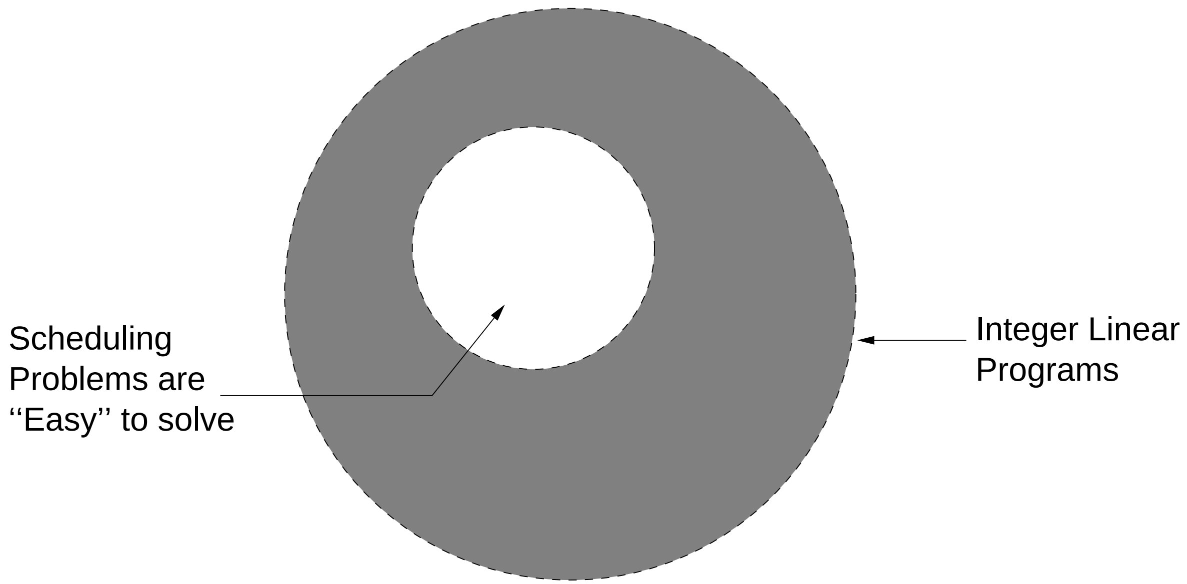
- Abstract representations.
- specifications are *decoupled* from program's control flow and syntax.
- Clean and minimal mechanism for expressing relationships.
- Integer-linear constraints are rich in their expressive power.
- Typically, abstract representations — Expensive compilation and analysis cost.

Feasibility via Analysis

- Constraint Satisfaction.
- In general, satisfiability of *Integer Linear Programs* — A hard problem.
- A key observations makes the *difference*.

All the Integer Linear Programs are Scheduling Problem!

More on Classical Scheduling



- Scheduling problems are extremely "well-solved".
- Fast algorithms produce very good solutions.
- Productions codes in industry run very well.

The Scheduling Model

In a *program dependence graph* representation

- *Data-* and *control-dependences* encode "precedence constraints".

- Instructions are the nodes.

- Pipelines delays on the target processor are latencies on the edges.

- A number and types of processor in the target processor are specified.

Traditional Well-solved Settings

- Acyclic control-flow structures.

- The goal is to minimize

  - completion time to improve performance of code on the RISC processor,

  - absolute deadlines to model limited form of hard real-time applications.

The Core Research Question

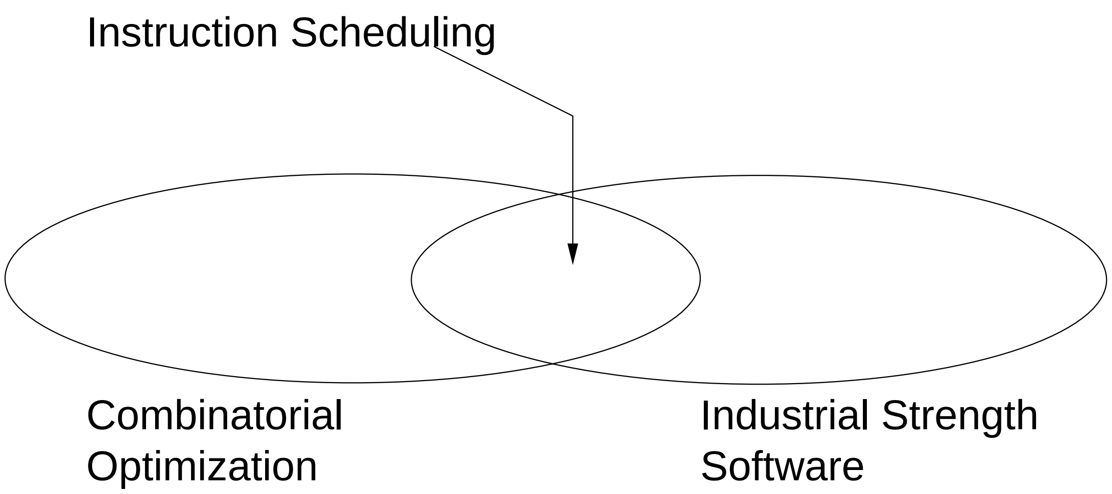
Reduced to solving a *Precedence-constrained Scheduling* problem with

- *cycles* in the graph,
- *relative* deadlines and start-times. As opposed to the absolute variants that have been addressed.

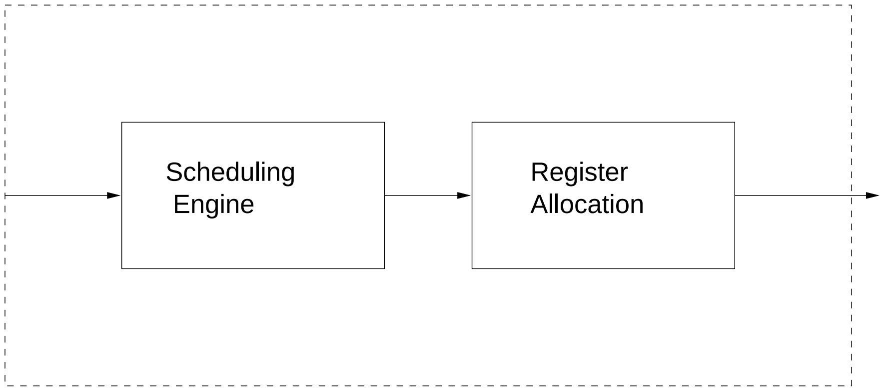
The entire problem is thereby reduced to a well-defined combinatorial optimization question.

Significance of Reduction

- A single well-parameterized combinatorial optimization question captures the essence of:
  - Analysis and
  - Code generation (to be seen)
- Amenable to
  - Powerful technical methods and
  - Software engineering approaches



The Plug

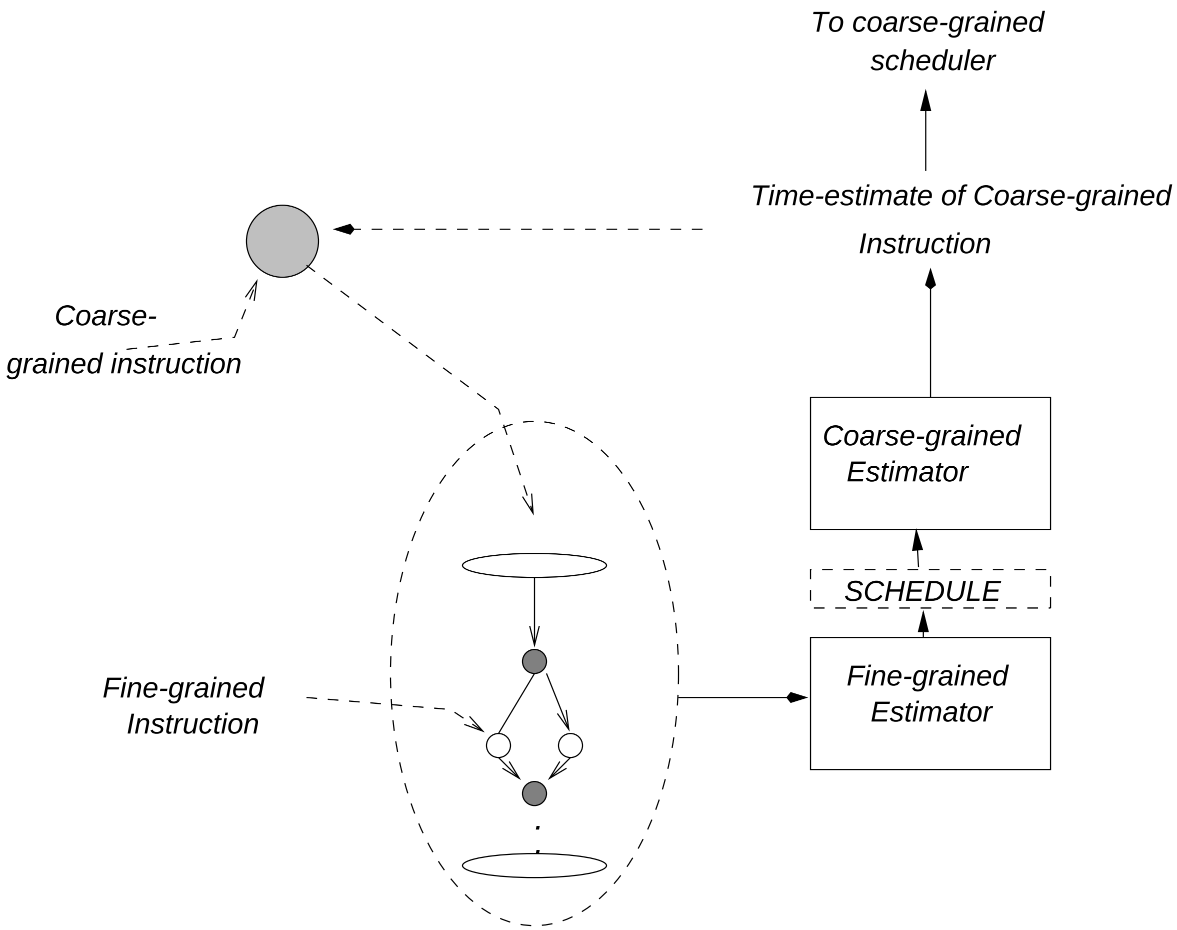


- Same scheduling engine for code generation (as for analysis).
- Corresponds to a closer relationship between the analysis and the code generation phases.
- Hence, the feasibility of compiled code is only and intimately dependent on the quality of the library describing the objects.
- Very good libraries are available in the field — very small gap between the two phases — tight design cycles.

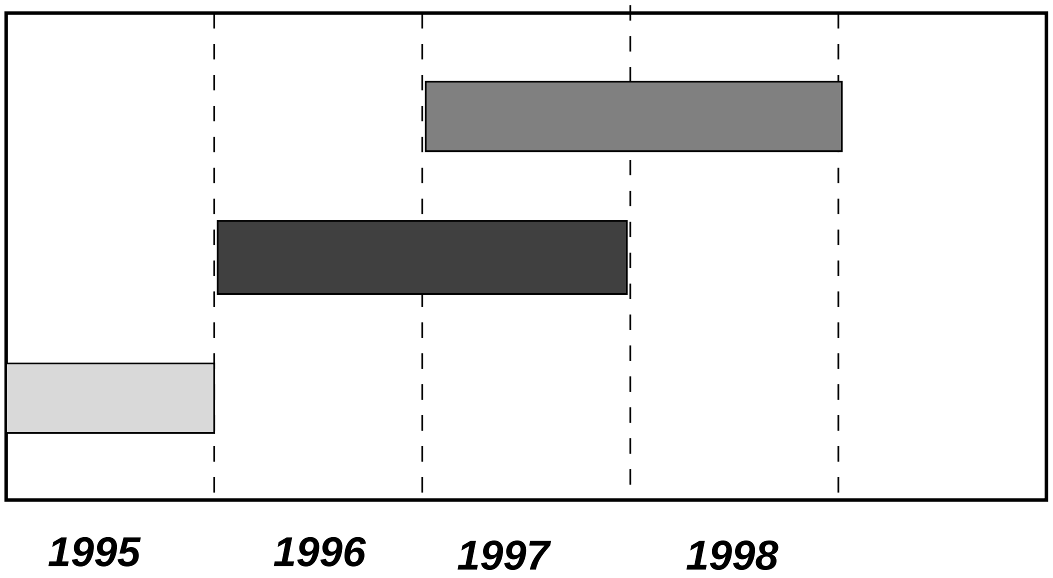
The Bonus

- In addition to aiding in the rapid-prototyping of real-time software, we are enabling the proliferation of RISC processors in the embedded system market.

Estimating Execution Times



Milestones



Prototyping Framework and Scheduling Engine Definition (Done)

Fine-grained Scheduling Algorithm Design and Evaluation

Develop Constraints Front-end and Integrate Scheduing Engine into Compiler, validate and Develop GUI (with N. Nachiappan)