Cours M2: Compilation avancée et optimisation de programmes

(transparents/slides un peu en français, a bit in English)

Alain Darte

CNRS, Compsys
Laboratoire de l'Informatique du Parallélisme
École normale supérieure de Lyon

Cours 1, 12 septembre 2012

Outline

Qu'est-ce que la compilation ?

- c'est le chaînon manquant qui relie le "software" au "hardware", le logiciel à l'architecture.
- très important pour comprendre ce qu'est un langage, un programme, un ordinateur, un circuit, etc.
- c'est le coeur de l'informatique et de l'automatisation.

Plusieurs aspects

- Traduction
- Optimisation ☛ sujet principal du cours
- Maintenance et généricité

Optimisations

- Agressive ou just-in-time
- Front-end ou back-end
- Algorithmique et/ou heuristique

Optimisations de codes et synthèse de circuits

- Adéquation entre architecture, compilation, applications.

Optimisations de codes et synthèse de circuits

- Adéquation entre architecture, compilation, applications.
- Frontière floue entre micro-processeur (programmable) et accélérateur matériel (non programmable).

Optimisations de codes et synthèse de circuits

- Adéquation entre architecture, compilation, applications.
- Frontière floue entre micro-processeur (programmable) et accélérateur matériel (non programmable).
- Besoin industriel. En France : Thales, STMicro, Kalray, ...

Optimisations de codes et synthèse de circuits

- Adéquation entre architecture, compilation, applications.
- Frontière floue entre micro-processeur (programmable) et accélérateur matériel (non programmable).
- Besoin industriel. En France : Thales, STMicro, Kalray, ...
- Tous les principes d'exécution (parallélisme, pipeline, mémoires distribuées, communications, threads) sont utilisés sur une même plate-forme (système embarqué).

Optimisations de codes et synthèse de circuits

- Adéquation entre architecture, compilation, applications.
- Frontière floue entre micro-processeur (programmable) et accélérateur matériel (non programmable).
- Besoin industriel. En France : Thales, STMicro, Kalray, ...
- Tous les principes d'exécution (parallélisme, pipeline, mémoires distribuées, communications, threads) sont utilisés sur une même plate-forme (système embarqué).
- Rapprochement entre synthèse de circuits (informatique et micro-électronique) et compilation (informatique et mathématiques). ☛ Ex : le CEA se met à faire du logiciel.

Thèmes abordés en cours

Représentations intermédiaires bas niveau

Control-flow graph, static single assigment, loop-forest

Ordonnancement d'instructions

DAG, pipeline logiciel : ILP, approximabilité, retiming

Allocation de registres

Complexité, graphes, optimisation combinatoire, ILP

Analyses, notamment polyédriques

Calcul de dépendances, de durée de vie, de transferts

Transformations

Fusion de boucles, pavage, contraction de tableaux

Applications

VLIW, synthèse FPGA, parallélisation automatique

Different types of loops

Fortran DO loops :

```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

C for loops :

```
for (i=1, i<=N, i++) {
  for (j=1, j<=N, j++) {
    a[i][j] = c[i][j-1];
    c[i][j] = a[i][j] + a[i-1][N];
  }
}
```

C for and while loops :

```
y = 0; x = 0;
while (x <= N && y <= N) {
  if (?) {
    x=x+1;
    while (y >= 0 && ?) y=y-1;
  }
  y=y+1;
}
```

Uniform recurrence equations

$\forall (i,j)$ such that $1 \leq i,j \leq N$

$$\begin{cases} a(i,j) = c(i,j-1) \\ b(i,j) = a(i-1,j) + b(i,j+1) \\ c(i,j) = a(i,j) + b(i,j) \end{cases}$$

Different types of loops (cont'd)

FAUST : real-time audio signal processing (for music)

```
random = +(12345) ~ *(1103515245);
noise = random/2147483647.0;
process = random/2 : @(10);
```

Equivalent meaning

$$\begin{cases} R(t) = 12345 + 1103515245 \times R(t-1) \\ N(t) = R(t)/2147483647.0; \\ P(t) = 0.5 \times R(t-10) \end{cases}$$

Different types of loops (cont'd)

FAUST : real-time audio signal processing (for music)

```
random = +(12345) ~ *(1103515245);
noise = random/2147483647.0;
process = random/2 : @(10);
```

Equivalent meaning

$$\begin{cases} R(t) = 12345 + 1103515245 \times R(t-1) \\ N(t) = R(t)/2147483647.0; \\ P(t) = 0.5 \times R(t-10) \end{cases}$$

Other high-level loop-based languages :

- Matlab
- Fortran90
- StreamIt
- High Performance Fortran
- C for high-level synthesis
- Languages for multi-cores

Different types of loops (cont'd)

FAUST : real-time audio signal processing (for music)

```
random = +(12345) ~ *(1103515245);
noise = random/2147483647.0;
process = random/2 : @(10);
```

Equivalent meaning

$$\begin{cases} R(t) = 12345 + 1103515245 \times R(t-1) \\ N(t) = R(t)/2147483647.0; \\ P(t) = 0.5 \times R(t-10) \end{cases}$$

Other high-level loop-based languages :

- Matlab
- Fortran90
- StreamIt
- High Performance Fortran
- C for high-level synthesis
- Languages for multi-cores

and many more ... where exploiting and optimizing loops (explicit or implicit loops) as well as array-like storage is fundamental.

Loops : repetitive structures

Loops : control structures that describe in a few lines of code a (possibly) large number of computations.

- Using loops make the code size smaller.
- A repetitive structure is exposed and can be exploited.
- Hot parts of codes where optimizations are needed.
- Large potential for optimizations (e.g., parallelism, memory).
- A loop can be optimized in a complexity that depends on the code size, not on the number of operations it describes.
- Loops can be parameterized : parametric optimizations, notion of asymptotic optimality, code generation with loops.
- Intermediate abstraction : low-level enough to be mapped to hardware, high-level enough to enable powerful analysis.

Optimizing compilers

Many constraints and/or difficulties for an optimizing compiler :

- Translation to be correct and fast : need to control complexity.
- The compiler does not "know" what the program computes : it can only focus on structure, not content.
- Parallelism can be hidden, so analysis are needed.
- A compiler phase is automatic : all cases have to be considered.
- Many phases in the whole translation, with interactions.
- The user may want to know what the compiler is doing.
- ...

Different types of loop optimizers

Compilation for (multi)-processors

Parallelization is only one phase of the process, it should not destroy other optimizations. Can generate codes that are not human-readable but should consider the whole program.

Semi-automatic compilation

Transformations should be user-friendly, understandable, or at least readable by the programmer. Transformations should be simple, expressed at high-level.

Compilation for high-level synthesis (HLS)

Depends if the program is optimized for the designer or for a HLS tool. Process controlled from code to circuit. Generation of "complex" codes possible. Can focus on one part of the code. Domain-specific, niche optimizations.

Research methodology : A side

Goal : improve performances of benchmarks on current platforms.

- Study some benchmarks by hand.
- Introduce new transformations for these particular examples.
- Integrate these techniques in the compiler.
- Justify the interest with experimental results on benchmarks.

Weaknesses :

- Needs a complete compiler or simulator.
- Insists only on positive cases.
- Is not explanatory (interface problem with the user).
- Benchmarks and performance curves arguable.

Research methodology : B side

Goal : study the power & limits of automatic code transformations.

- Try to classify the problems : NP-completeness, guaranteed heuristics, complete methods, polynomial algorithms, etc.
- Help to clean up the literature. Try to not reinvent the wheel.
- Define a model in which a method can be applied : minimal hypotheses and counter-examples that prevent generalization.
- Abstract properties, avoid discussions on particular cases.

Weaknesses

- Explains but does not solve practical situations.
- Hard to evaluate the pertinence of problems and models.
- May miss interactions with other optimizations.

Theoretical and practical loop problems

There is a huge range of problems & techniques related to loops.

Analysis

- Dependence & lifetime analysis.
- Abstract interpretation.
- Induction variable recognition.
- Cache access pattern analysis.
- Program termination.

Scheduling

- Software pipelining.
- Circuit retiming.
- Parallelism detection.
- Vectorization.
- Communication optim.

Transforming

- Memory reuse : array privatization & contraction.
- Locality optimizations : spatial and temporal reuse.
- Unimodular transformations, fusion/distribution, tiling, etc.
- Code generation with or without loop counters.

Example : software pipelining, 1D cyclic problem

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
  ld[r26], r27
    nop
  add r27, 6740, r26
  ld 0x1A54[r27], r27
    nop
  sub.f r27, r25, r0
  bne L400
    nop
L399 :
```

$8n$ cycles.

Example : software pipelining, 1D cyclic problem

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code	Code compaction
L400 : **ld**[r26], r27 nop **add** r27, 6740, r26 **ld** 0x1A54[r27], r27 nop **sub.f** r27, r25, r0 **bne** L400 nop L399 :	L400 : **ld**[r26], r27 nop **ld** 0x1A54[r27], r27 **add** r27, 6740, r26 **sub.f** r27, r25, r0 **bne** L400 nop L399 :
$8n$ cycles.	$7n$ cycles.

Example : software pipelining, 1D cyclic problem

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code	Code compaction	Sofware pipelining
L400 : **ld**[r26], r27 nop **add** r27, 6740, r26 **ld** 0x1A54[r27], r27 nop **sub.f** r27, r25, r0 **bne** L400 nop L399 :	L400 : **ld**[r26], r27 nop **ld** 0x1A54[r27], r27 **add** r27, 6740, r26 **sub.f** r27, r25, r0 **bne** L400 nop L399 :	**ld**[r26], r27 nop **add** r27, 6740, r26 L400 : **ld** 0x1A54[r27], r27 **ld**[r26], r27 **sub.f** r27, r25, r0 **bne** L400 **add** r27, 6740, r26 L399 :
$8n$ cycles.	$7n$ cycles.	$3 + 5n$ cycles.

Network processor Myricom LANai 3.0

- Une seule unité pipelinée de profondeur 3 (ou 4 selon la version), mais le délai apparent d'une instruction est :
  - 1 pour moves ou opération sur ALU.
  - 2 pour jumps ou load/store ☛ ombre ("shadow").
- Une seule possibilité de "control hazard" :

  r1 = load(toto)
  r1 = r2 + 1

  C'est l'opération sur les registres (la seconde) qui a priorité.
- Pas de hiérarchie mémoire.

Agere Payload Plus

- VLIW de largeur 4.
- 1 ALU par "slice" de 1 octet.
- 32 registres de 4 octets, un pour chaque "slice", plus registres spéciaux Y et Q :
  - Q = mémoire temporaire, "slice" par "slice" ou décalage global.
  - Y = écrit "slice" par "slice", lu par tout le monde.
    ☛ Seul moyen de communication !
  - Code à . . . 2 adresses (sauf pour Q et Y) !

Ubicom IP3023

- Processeur scalaire "in order".
- 8 contextes de registres pour support "multi-thread".
- Possibilité de mélanger les instructions de différents "threads" cycle par cycle.
- 256Ko I-scratchpad + 64Ko D-scratchpad.

ST220 family

- Processeur VLIW (largeur 4).
- Direct-mapped I-Cache (512 lignes de 64o).
- Prédication limitée (select = move conditionnel).
- Multiply-add, auto-incréments.

Registres et instructions spécialisés

- Registres d'adressage avec incréments et décréments. Ex : TI TMS320C25, Motorola DSP56K
- Registres rotatifs. Ex : Itanium
- "Clusters" de registres. Ex : Agere Payload
- "Register windows". Ex : WMIS Microcontroller
- Registres pour load/store doubles. Ex : IBM PowerPC405, Intel StrongARM pour IXP-1200
- Extensions SIMD (MMX puis SSE) et DSP extensions. Ex : AMD, Pentium, XScale, ARM, etc.

Support matériel pour l'exécution

- Cache d'instructions
- Scratchpad pour les instructions et les données
- Support pour l'exécution des boucles
- Compression de code. Ex : Atmel Diopsis Dual Core DSP
- Processeurs extensibles (custom FUs). Ex : Xtensa (Tensilica)
- Puissance : drowsy modes, ...

Plus qu'un processeur

GPU (Graphics processing unit) contraintes fortes de mémoire locale et de transferts de données.

Carte FPGA mélange de processeurs programmables et d'accélérateurs matériels.

Multi-cœurs processeurs homogènes (en général), accélérateurs matériels, mémoires et caches partagés (avec cohérence de divers types).

☛ Recherche en :

- vérification de code
- certification de compilateur
- langages
- compilateurs
- systèmes d'exploitation et multi-threading
- architecture

Outline

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

$8n$ cycles.

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

$8n$ cycles.

Code compaction

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

$8n$ cycles.

Code compaction

```
L400 :
   ld[r26], r27
      nop
   ld 0x1A54[r27], r27
   add r27, 6740, r26
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code	Code compaction
L400 : **ld**[r26], r27 nop **add** r27, 6740, r26 **ld** 0x1A54[r27], r27 nop **sub.f** r27, r25, r0 **bne** L400 nop L399 :	L400 : **ld**[r26], r27 nop **ld** 0x1A54[r27], r27 **add** r27, 6740, r26 **sub.f** r27, r25, r0 **bne** L400 nop L399 :
$8n$ cycles.	$7n$ cycles.

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code	Code compaction	Sofware pipelining
L400 : **ld**[r26], r27 nop **add** r27, 6740, r26 **ld** 0x1A54[r27], r27 nop **sub.f** r27, r25, r0 **bne** L400 nop L399 :	L400 : **ld**[r26], r27 nop **ld** 0x1A54[r27], r27 **add** r27, 6740, r26 **sub.f** r27, r25, r0 **bne** L400 nop L399 :	L400 : **ld**[r26], r27 nop **add** r27, 6740, r26 **ld** 0x1A54[r27], r27 nop **sub.f** r27, r25, r0 **bne** L400 nop L399 :
$8n$ cycles.	$7n$ cycles.	

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400 :
   ld[r26], r27
      nop
   add r27, 6740, r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

$8n$ cycles.

Code compaction

```
L400 :
   ld[r26], r27
      nop
   ld 0x1A54[r27], r27
   add r27, 6740, r26
   sub.f r27, r25, r0
   bne L400
      nop
L399 :
```

$7n$ cycles.

Sofware pipelining

```
      ld[r26], r27
         nop
      add r27, 6740, r26
L400 :
      ld 0x1A54[r27], r27
         nop
      sub.f r27, r25, r0
      bne L400
      ld[r26], r27
         nop
      add r27, 6740, r26
L399 :
```

Software pipelining : example for LANai 3.0

Optimize throughput, with dependence & resource constraints.

Ex : sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code	Code compaction	Sofware pipelining
L400 : **ld**[r26], r27 nop **add** r27, 6740, r26 **ld** 0x1A54[r27], r27 nop **sub.f** r27, r25, r0 **bne** L400 nop L399 :	L400 : **ld**[r26], r27 nop **ld** 0x1A54[r27], r27 **add** r27, 6740, r26 **sub.f** r27, r25, r0 **bne** L400 nop L399 :	**ld**[r26], r27 nop **add** r27, 6740, r26 L400 : **ld** 0x1A54[r27], r27 **ld**[r26], r27 **sub.f** r27, r25, r0 **bne** L400 **add** r27, 6740, r26 L399 :
$8n$ cycles.	$7n$ cycles.	$3 + 5n$ cycles.