PIPS-related projects at TÉLÉCOM Bretagne

Ronan KERYELL rk@enstb.org

Institut TÉLÉCOM, TÉLÉCOM Bretagne, Département Informatique,
Lab-STICC/CAC/HPCAS
Plouzané, France

22 février 2008

Le plan

Le plan

Introduction (I)

Computer Science department at TÉLÉCOM Bretagne involved in the SAFESCALE project :

- Refining secure high-performance processor : CryptoPage
- Simulating this processor
- Writing a parallelizing tool to transform classical applications for the SAFESCALE architecture based on Kaapi execution model

Le plan

SAFESCALE execution model (I)

- Parallel applications describes as a task graph
- Communicating tasks with a data-flow model with consumer-producer paradigm
- SAFESCALE adds secure execution through different mechanisms (from more to less efficient)
  - Use inherently fault tolerant algorithm if possible (fix point, iterative solvers...)
    - Open question : how many applications are in this category ?
  - Secure execution platform (if available)
    - Trust the microprocessor factory...

SAFESCALE execution model (II)

- Right now, only “smartcard supercomputers” (LaBRI) or military computing platforms (SAGEM Fox...) ☹

- ▶ Use fault tolerant property to component based-applications
- ▶ Replay few tasks on some more (“verifiers”) or less secure nodes to verify they produce the same results
  - Use a probabilistic approach to grant a probabilistic security level
  - Use the dependence graph to optimize task verification selection
- ▶ Use the BCP (*Best Current Practices*) to secure execution in *the* real world

Writing SAFESCALE applications (I)

- Write application as a task DAG
- Task DAG ≈ SSA for imperative programs
  - If loops with task calls in them, need to unroll them...
  - May not be quite convenient for average programmers ☺
- Need to study more deeply Kaapi model and API
- Need to define a SAFESCALE API (Kaapi is a good starting point ☺)
- A tool to help writing SAFESCALE applications could be nice...

Automatic compilation for SAFESCALE (I)

- Real programmers love big sequential machines ☺
- Need to split a big application in several tasks...
- ...if tasks are parallel, it is better ☺
- Need to figure out
  - Data needed by a task to run
  - Data produced by a task used later by other tasks to run
- Add some Kaapi/SAFESCALE glue to
  - Define and calling the tasks
  - Map the task inputs to actual variables used in each task

Automatic compilation for SAFESCALE *(II)*

  - Map the task variable productions to task output
- Orchestrate all these tasks to preserve the initial program semantics

Idea : use the PIPS source-to-source compiler from ENSMP to automate this process

Code distribution (I)

- General problem already studied and going on at ENSMP, ENSTB and in many other places
  - Compilation for SoC with hardware accelerators : CoMap project with UBO & ENSTB
    - Extract pieces of code in an application
    - Compile them in hardware accelerators
    - Add some glue to interface to the sequential program (bus, bus API, DMA generation...)

    PhD thesis at UBO and MSc of Matthieu GODET
  - Poor man OpenMP : INT with ENSTB
    - Offer distributed shared memory computing without... shared memory ! ☺

Code distribution *(II)*

  - Compute more or less the memory zones used as input and output
  - Mimic the shared memory semantics with different API (remote write, MPI...)
- Mobython at INT, CNAM and ENSTB
  - Build a grid on cellular phones ☺
  - Transform an application into distributed tasks
- HPF compilation : Jean-Louis PAZAT ☺, ENSMP
- SAFESCALE !

- Idea $\rightsquigarrow$ factorizing these transformations in PIPS
- MSc of The Nhan LUONG and 1-year post-doc to be found for SAFESCALE... Done ! ☺

Distribution concepts in PIPS (I)

- Many semantics analysis and code transformations available in PIPS
- Parallelization transformations usable here to extract parallel tasks
- Interprocedural dependence graph usable to feed the Kaapi scheduler
- PIPS compute “Regions” that define storage areas used by a piece of code : use them to generate communications
- Parallelization is in the general case not decidable : need to think about SAFESCALE directives to help the compiler too

PIPS read and write array regions (I)

- Array elements described as integer polyhedra
- Integer polyhedra : compromise between expressivity and easy mathematical management
- Not all the memory accessed can be sum up with polyhedra
- $\leadsto$ Array regions can be exact (the elements accessed are exactly described) or inexact (with more points than really accessed, to be conservative)
- Regions are built up bottom-up through hierarchical statements

PIPS read and write array regions (II)

```
//␣N-r-exact{}
double␣b[N],␣a[N];␣int␣i;␣...

//␣i-w-exact{}
//␣i-r-exact{}
//␣N-r-exact{}
//␣a[$\varphi_1$]-r-exact{$0 \leq \varphi_1 \bigwedge \varphi_1 \leq N$}
//␣b[$\varphi_1$]-r-inexact{$0 \leq \varphi_1 \bigwedge \varphi_1 \leq N$}
//␣Code␣to␣execute␣somewhere␣else:
{
␣␣//␣s-w-exact{}
␣␣double␣s␣=␣0;
␣␣//␣i-w-exact{}
␣␣//␣i-r-exact{}
␣␣//␣N-r-exact{}
```

PIPS read and write array regions (III)

```
␣␣//␣a[φ1]-r-exact{0 ≤ φ1 ⋀ φ1 ≤ N}
␣␣//␣b[φ1]-r-inexact{0 ≤ φ1 ⋀ φ1 ≤ N}
␣␣for(i␣=␣0;␣i␣<␣N;␣i++)␣{
␣␣␣␣//␣i-r-exact{}
␣␣␣␣//␣a[φ1]-r-exact{φ1 == i}
␣␣␣␣//␣s-r-exact{}
␣␣␣␣//␣s-w-exact{}
␣␣␣␣s␣+=␣a[i];
␣␣␣␣//␣s-r-exact{}
␣␣␣␣//␣i-r-inexact{}
␣␣␣␣//␣a[φ1]-r-inexact{φ1 == i}
␣␣␣␣//␣b[φ1]-w-inexact{φ1 == i}
␣␣␣␣if␣(s␣>␣0)
␣␣␣␣␣␣//␣i-r-exact{}
␣␣␣␣␣␣//␣a[φ1]-r-exact{φ1 == i}
```

PIPS read and write array regions (IV)

```
      // b[φ1]-w-exact{φ1 == i}
      b[i] = 2*a[i];
  }
}
// b is used later
...
```

Read and write regions are overkill for us because some statements may write elements that are not used later. . .

PIPS in and out array regions (I)

- Use the dependence graph to compute elements that are *really* used by a statement (*in* regions) and that are written and will *really* be needed by a future statement (*out* regions)

```
//␣N-in-exact{}
double␣b[N],␣a[N];␣int␣i;␣...

//␣N-in-exact{}
//␣a[$\varphi_1$]-in-exact{$0 \leq \varphi_1 \bigwedge \varphi_1 \leq N$}
//␣b[$\varphi_1$]-out-inexact{$0 \leq \varphi_1 \bigwedge \varphi_1 \leq N$}
//␣Code␣to␣execute␣somewhere␣else:
{
␣␣//␣s-out-exact{}
```

PIPS in and out array regions *(II)*

```
␣␣double␣s␣=␣0;
␣␣//␣N-in-exact{}
␣␣//␣a[$\varphi_1$]-in-exact{$0 \leq \varphi_1 \bigwedge \varphi_1 \leq N$}
␣␣//␣b[$\varphi_1$]-out-inexact{$0 \leq \varphi_1 \bigwedge \varphi_1 \leq N$}
␣␣//␣s-in-exact{}
␣␣for(i␣=␣0;␣i␣<␣N;␣i++)␣{
␣␣␣␣//␣i-in-exact{}
␣␣␣␣//␣a[$\varphi_1$]-in-exact{$\varphi_1 == i$}
␣␣␣␣//␣s-in-exact{}
␣␣␣␣//␣s-out-exact{}
␣␣␣␣s␣+=␣a[i];
␣␣␣␣//␣s-in-exact{}
␣␣␣␣//␣i-in-inexact{}
␣␣␣␣//␣a[$\varphi_1$]-in-inexact{$\varphi_1 == i$}
␣␣␣␣//␣b[$\varphi_1$]-out-inexact{$\varphi_1 == i$}
```

PIPS in and out array regions (III)

```
    if (s > 0)
      // i-in-exact{}
      // a[φ1]-in-exact{φ1 == i}
      // b[φ1]-out-exact{φ1 == i}
      b[i] = 2*a[i];
  }
}
// b is used later
...
```

SAFESCALE generation blueprint (I)

To distribute a statement $\mathcal{S}$ on a node $\mathcal{N}$:

- $\texttt{receive}_{\mathcal{N}}(e \in \mathrm{InExact}(\mathcal{S}) \bigcup \mathrm{InInexact}(\mathcal{S}))$
- $\texttt{receive}_{\mathcal{N}}(e \in \mathrm{OutInexact}(\mathcal{S}))$
- $\texttt{executeTask}_{\mathcal{N}}(\mathcal{S})$
- $\texttt{send}_{\mathcal{N}}(e \in \mathrm{OutExact}(\mathcal{S}) \bigcup \mathrm{OutInexact}(\mathcal{S}))$

Optimized SAFESCALE generation *(I)*

- For inexact out regions, may use combining write instead of read-then-write
  - Need to track modified elements with a run-time resolution
  - May use inspector-executors
  - Need to detect loop invariant region patterns
- In the generation sketch up, guess that variables are allocated in the same way on each task using them
  - Quite inefficient if a task use only few element
  - Use a more compact allocation
- What about the general pointers ?

Optimized SAFESCALE generation (II)

- Some science fiction : reorganize globally variables for more efficient communications and data access (spitting red-black relaxation data into red and black in 2 different arrays. . .)
- Big collaboration needed here in SAFESCALE : everything still to do ☺

Le plan

Conclusion (I)

- CryptoPage architecture developing at good pace with PhD and MSc students
- Good simulation results : only 3 % slowdown on average by combining HIDE, fast cipher at page level and speculative insecure execution (presentation at ACSAC 12/2006)
- Interesting side effects : need SAFESCALE to simulate ~~SAFESCALE~~CryptoPage ☺
- Still looking for a (good ☺) 1-year post-doc to work on compilation aspect with $\leadsto$ cold start for this part of the project with IMAG team ☹

Conclusion (II)

- Factorizing compilation aspects for other heterogeneous parallel machines

Le plan

Projet PHRASE

Générer code pour acélérateurs configurables

- Détourage de code marqué par directives qui est déporté sur accélérateur
- Atomiseur de code 3 adresses
- Génération bus logiciel/matériel
- Transformation graphe de contrôle en FSM hiérarchique ou à plat
- Prettyprinter SmallTalk pour outil de synthèse Madeo du Lab-SSTIC/CAC/AS

Projet SAC

SIMD Architecture Compiler

- Vectoriseur par déroulage et rassemblement d'instructions « en vrac »
- Calcul taille opérandes à partir des précondition
- Génération d'intrinsèques à partir fichier configuration architecture
- Gère réduction

Projet CoMap

- Détourage de code marqué par directives
- Génération de descripteur DMA et des synchronisations
- Génération code de contrôle
- Génération du code des opérateurs

Le plan

NewGen

Échanger données avec autres outils ?

- Rajouter un backend XML
- Regarder STEP/EXPRESS ISO 10303 pour échanger données ?

La syntaxe ne fait pas tout... Quid de la sémantique ☹

Interfaces graphiques

Remettre en ordre...

- Jpips (au dessus de tpips) devrait encore marcher ?
- wpips utilise XView plus maintenu ☹
- epips utilise GNU/Emacs mais au dessus de wpips

Epips dans Eclipse ?

Rajouter snapshot dans PIPSdbm pour permettre de défaire des actions

PIPS & checkpoint

Problème de robustesse des machines à moult processeurs

Example

BlueGene EPFL : 1 panne/2,17 jours

- Contact avec le LRI sur snapshot/checkpoint
- Collaboration avec TÉLÉCOM SudParis qui travaille dans le domaine ?

Relations avec IEF *(I)*

- IEF QUAFF génère code à partir de templates C++ pour programmes basés sur squelettes algorithmiques
- Utilisation de PIPS
  - Front-end : génère programmes QUAFF
  - Back-end/Middle-end : optimise/transforme sortie de QUAFF

Relations avec IEF *(II)*

- Question : peut-on récupérer C++ détemplatisé sans génération de code ? Comment générer du C++ (compilable par la suite y compris avec des templates) plutôt que de l'objet ?
  Si oui, permettrait d'utiliser QUAFF comme middle-end moult simplement pour moult cibles
  - Langages StreamC, RapidMind, CUDA, HMPP (CAPS Entreprise)
  - Bibliothèques AMD FrameWave, Intel Threading Building Blocks, KAAPI (LIG)
  - MPI (SKELL BE)
  - OpenMP, FORESTGOMP (LaBRI)

Relations avec IEF *(III)*

- Implémentations de C++ STL PaSTeL (multi-cœurs, LIG), MCSTL (Karlsruhe), STXXL, STAPL...
- FREIA : templates pour C++ ou SystemC synthétisable
- PIPS en post-production : rajouter des annotations éventuellement pour raffiner avec PIPS en post-production ?

À rajouter/faire dans PIPS (I)

- Terminer passage en 64 bits
- Refaire un site WWW moderne
  - Utiliser un CMS collaboratif, style bibliographie interactive `hpcas.enstb.org/resources-related-projects`
- Généralisation de la transformation de programmes
- Utiliser langage de réécriture de graphes ? Pattern matching dans PIPS ?
- Tiling généralisé et paramétrable
- Pipeline logiciel

À rajouter/faire dans PIPS *(II)*

- Généralisation factorisation sous-expressions communes
- Faire état des lieux de tout ce qui a été fait et refactoriser

Conclusion

- Moult choses disponibles dans PIPS
- Moult choses encore à faire
- Projet toujours vivant au bout de 20 ans ! ☼
- Gros travail de public relation à faire...
- Organiser fiesta pour les 20 ans de PIPS ! ☺

Table des matières