Software Pipelining and Register Pressure in VLIW Architectures: Preconditionning Data Dependence Graphs is Experimentally Better than Lifetime-Sensitive Scheduling

Frédéric Brault, Benoît Dupont-de-Dinechin, Sid-Ahmed-Ali Touati, Albert Cohen

ODES 2010
An old debate about an open question

- Phase ordering problem: instruction scheduling before/after register allocation?
- Highlighted in the 80’s for sequential code, with register minimisation
- Wealth of heuristics for acyclic scheduling
- What about cyclic scheduling?
Related work

- Software pipelining under resource constraints only
  → register pressure often goes out of control

- Software pipelining under resource and register constraints
  → to spill or to increase the II – that is the question

- Post-pass cyclic register allocation
  → necessary: modulo expansion (unrolling) and register assignment
Our strategy for VLIW

1. Decoupling register pressure control from instruction scheduling
   → better compiler engineering
   → focus scheduling on the core objectives (Ⅱ, hiding memory latency)

2. Handling register constraints before scheduled resource constraints
   → Memory operations have unknown static latencies → Imprecise scheduling and WCET analysis

3. Avoid spilling instead of scheduling spill code while taking care of Ⅱ
   → Memory operations consume more power
The target platform

**ST231 processor**
- 4-issue VLIW processor at 400 MHz
- 64 general purpose 32-bit registers (GR)
- 8 1-bit condition registers (BR)
- 1 LSU, 1 BCU, 4 ALU and 1 MAU functional units
- 32 KB 4-way Dcache, 32 KB direct-mapped Icache

**Toolchain: ST200cc with LAO**
- Front-end compiler based on Open64
- At -O3 optimization level, the LAO backend component performs VLIW software pipelining
- Post-pass register allocation in ST200cc
SIRA: an example

(a) Initial DDG

(b) Reuse Graphs for Register Types $t_1$ and $t_2$

(c) DDG with Killing Nodes

(c) $V^k = \{k_{u^t} | u \in V^{R,t}\}$

(c) $E^k = \{(v, k_{u^t}) | v \in Cons(u^t)\}$

(d) Preconditioned DDG after Applying SIRA

(d) $E^\mu = \{(k_{u^t}, v) | (u, v) \in E^{reuse,t}\}$
Comparing SIRA vs. existing work

- **Unique features of SIRA**
  - Optimise for multiple register types simultaneously or one after another
  - Model (read and write) delays in accessing registers
  - Model register banks, buffers or rotating register files.
  - Register pressure guarantee independent of the scheduling algorithm
  - Correctness proofs for the model and algorithms
  - Reproducible results: standalone C library (SIRAlib), distributed with experimental data

- **Validation of the effectiveness of SIRA in a production compiler**
  - Compiler construction: simplifies scheduling/allocation ordering
  - Software engineering: SIRA as an independent C library plugable in any compiler
  - Reproducibility: the source code is publicly released (LGPL)
  - Effectiveness: already published for standalone DDG, experimental results of this talk for an integrated context.
SIRA: schedule independent register allocation

Fundamental principle: Theorem [Touati2001]

Let $G$ be a loop DDG. Let $G'$ the extended DDG of $G$ associated with the valid reuse graph $G^{\text{reuse},t}$ for the register type $t$. Then, any software pipelining $\sigma$ of $G$ does not require more than $\sum \mu_{u,v}^t$ registers of type $t$, where $\mu_{u,v}^t$ is the reuse distance between $u$ and $v$ in $G^{\text{reuse},t}$. Formally:

$$\forall \sigma \in \Sigma(G), \text{PeriodicRegisterRequirement}^t_\sigma(G) \leq \sum \mu_{u,v}^t$$
SIRA: How it works?

The SIRALINA heuristic works in two polynomial steps:

1. **Step 1:** Computes the minimal reuse distances between every possible couple of statements (i.e. Compute a function $\mu^t : V^R_t \times V^R_t \rightarrow \mathbb{Z}$ for each register type $t$);

2. **Step 2:** Compute a bijection $E^{reuse,t} : V^R_t \rightarrow V^R_t$ that minimises $\sum_{e_r \in E^{reuse,t}} \mu^t(e_r)$ for each register type $t$. 
SIRA: How it works?

Step 1: It is a cyclic scheduling problem under precedence constraints only. It may be solved optimally by a min-cost max flow problem, or by a linear program with a totally unimodular constraints matrix. The complexity is $O(||V||^3 \log ||V||)$.

Step 2: It is a linear assignment problem, solved optimally by the Hungarian algorithm in $O(||V||^3)$. 
SIRA: How it works?

(a) Initial DDG

(b) Reuse Graphs for Register Types $t_1$ and $t_2$

(c) DDG with Killing Nodes

\[ V^k = \{k_{u^t} | u \in V^{R,t} \} \]

\[ E^k = \{(v, k_{u^t}) | v \in Cons(u^t)\} \]

(d) Preconditioned DDG after Applying SIRA

\[ E^\mu = \{(k_{u^t}, v) | (u, v) \in E^{reuse,t} \} \]
Plugging SIRA into the ST231 toolchain

1. **C file**
   - st200cc compiler (high level optimisations)
   - WHIRL representation
2. **LAO backend optimisation** (superblock formation, advanced global VLIW scheduling, SWP)
3. **DDG**
   - Modify the DDG with SIRA (bound MAXLIVE for any subsequently SWP)
   - Optimal SWP with integer programming
   - Unwinding SWP (resource constraints)
   - Lifetime-sensitive SWP
4. **st200cc postpass** (register allocation, target specific optimisation)
5. Assembly VLIW
Experiments

Setup
- FFMPEG, MEDIABENCH and SPEC CPU2000 benchmarks
- ST231 register count lowered to 32 GR, 4 BR, optimized simultaneously

Instruction schedulers
- SIRA frees aggressive scheduling from register pressure worries
  1. Optimal: Integer Linear Programming, minimize II and schedule length
  2. Unwinding heuristic: unrolling-based method to build modulo schedules
  3. Lifetime-sensitive heuristic: minimizes the sum of life-ranges

Questions
- Does SIRA improve performance? For which scheduler?
- How does a lifetime sensitive heuristic compare with the combination of SIRA with a pressure-unaware algorithm?
Experiments

Setup

- Instrumentation of the toolchain yields static numbers about spills and II.
- For each benchmark and each scheduler, we compare the numbers obtained with the scheduler alone to those obtained with both SIRA and the scheduler.
Experiments

- Mean spill variation = \[ \frac{\sum (\text{Spill}_{\text{with.SIRA}} - \text{Spill}_{\text{without.SIRA}})}{\sum \text{Spill}_{\text{without.sira}}} \]

- Mean II variation = \[ \frac{\sum (\text{II}_{\text{with.SIRA}} - \text{II}_{\text{without.SIRA}})}{\sum \text{II}_{\text{without.SIRA}}} \]
Experiments: cross-comparison

**Question**
How does a lifetime sensitive heuristic compare with the combination of SIRA with a pressure-unaware algorithm?

**Setup**
- SIRA + unwinding scheduler vs. lifetime-sensitive scheduler alone
- SIRA + optimal scheduler vs. lifetime-sensitive scheduler alone
Experiments: cross-comparisons

SIRA + scheduler vs Lifetime-sensitive

Spill decrease in %

FFMPEG  MEDIABENCH  SPEC2000

SIRA + scheduler vs Lifetime-sensitive

II decrease in %

FFMPEG  MEDIABENCH  SPEC2000
Experiments: spill code in post-pass

Does SIRA reduce spill or prevent it altogether?

Answer: evaluate \( \frac{\text{Loops that do not have spill anymore once SIRA is used}}{\text{Loops that had spill without SIRA}} \)

SIRA reduces the number of loops that have spill code
Conclusions

- Using SIRA significantly decreases both II and spills, for all schedulers.
- Not surprisingly, results are less impressive on the lifetime-sensitive scheduler, since the heuristic already reduce register pressure.
- The combination of SIRA with an aggressive scheduler outperforms the lifetime-sensitive approach.
The speedup debate

- Speedups depend on the data input, and the time fraction spend in the SWP loops.
- The compiler optimises for an architectural objective, while speedup comes from a complex interaction with the micro-architecture and the experimental environment.
- If you get a speedup, who guarantees that it comes as a direct consequence of the plugged optimisation? Phase ordering, hidden side effects, etc.
- In our case: SWP loops account for 0% to 5% of the whole application execution times. Most of the speedups are equal to 1.
- The other speedups vary from 0.85 to 2.4. Except in one case (FFMPEG), all the observed speedups and slowdons come from I-cache effects!
- Do not trust speedups when you work on code optimisation! Trust what you can prove or demonstrate, not what you observe. Code quality is a matter of many metrics, speedup is a single metric among many others.