A framework to experiment optimizations for real-time and embedded software

Hugues CASSÉ[1]
Karine HEYDEMANN[2]
Haluk OZAKTAS[2]
Jonathan PONROY[3]
Christine ROCHANGE[1]
Olivier ZENDRA[3]

[1]IRIT - University of Toulouse
[1]LIP6 - University Pierre and Marie Curie
[3]INRIA/LORIA Nancy

20 May 2010

Current Section

Real-time embedded systems

- Embedded systems constraints
  - Limited memory ⇒ limited code size
  - Possible need for autonomy and/or inability to use efficient but voluminous cooling equipments ⇒ low energy consumption
- Real-time applications
  - Hard or soft timing deadlines

⇒ Various optimizations must be applied in order to meet requirements!

- ANR granted MORE Project: **M**ulti-criteria **O**ptimizations for **R**eal-time **E**mbedded Systems

Code transformations

- Optimizations through code transformations to meet system requirements
  - Code size requirements
    - Compiler optimizations to reduce the number of instructions
    - Code compression
  - Energy requirements
    - Compiler transformations to reduce the number of memory accesses
    - Data placement strategies to optimize the use of various memories of the hardware
  - Worst-case execution time (WCET) requirements
    - Different kinds of code transformations, like loop unrolling or function inlining, aiming to remove flow control instructions

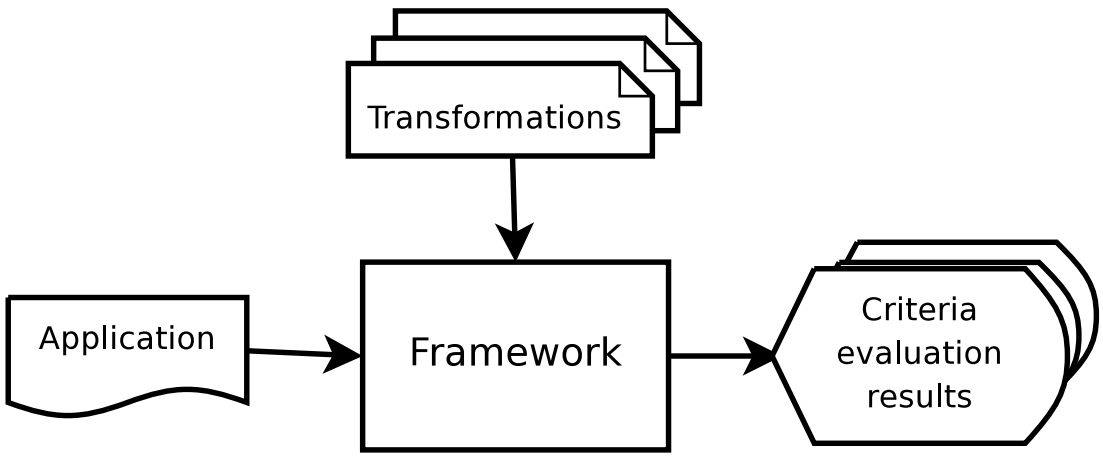
Side effects of transformations

- A sole transformation can have effects on several criteria
  - Code compression targets code size but can impact energy consumption, average and worst case execution times
  - Data placement strategies target energy consumption but can also impact average and worst case execution times
  - WCET related code transformations can also impact code size and energy consumption

$\Rightarrow$ To achieve a multi-criteria optimization, we must carefully analyze the effects of multiple transformations

A common framework

- A **common** framework for all transformations and evaluations is the key



  - Allows rapid and easy experimentation
  - Reduces time to market

Current Section

OTAWA: Open Tool for Adaptive WCET Analysis

- OTAWA: A framework dedicated to worst-case execution time analysis, including
  - tools to load and decode binary codes (PowerPC, ARM7, TriCore, Star12X)
  - tools to build a representation of the code (CFG: Control Flow Graph)
  - code processors that allow easy implementation of static code analyses
  - annotation facilities that allow hooking attributes (e.g. results of analyses/transformations) to any code object
  - cycle level simulator built on top of SystemC
    - supports superscalar pipelines, in-order and out-of-order execution, branch prediction, instruction and data caches, scratchpad memories, user-specified memory architecture...

Current Section

Energy-aware memory mapping

- Take advantage of different behaviors of the heterogeneous memories with respect to energy
  - e.g. main memory, cache memory, scratchpad memory (SPM)
  - Place the most accessed data in the less energy-hungry memories
    - SPM is targeted for its efficiency in energy consumption
- 3 strategies of static date placement:
  - *SPM_firstUsed*, *SPM_smallSizeFirst*, *SPM_highFrequency*
  - Static data placement avoids interacting with code compression and improves timing predictability
- Implementation of the transformation
  - A pre-run is performed to trace accesses to each piece of data
    - Added probes to `OTAWA` simulator
  - The trace is used to perform data mapping prior to actual run
    - New memory mapping is emulated in `OTAWA` by working on data accesses

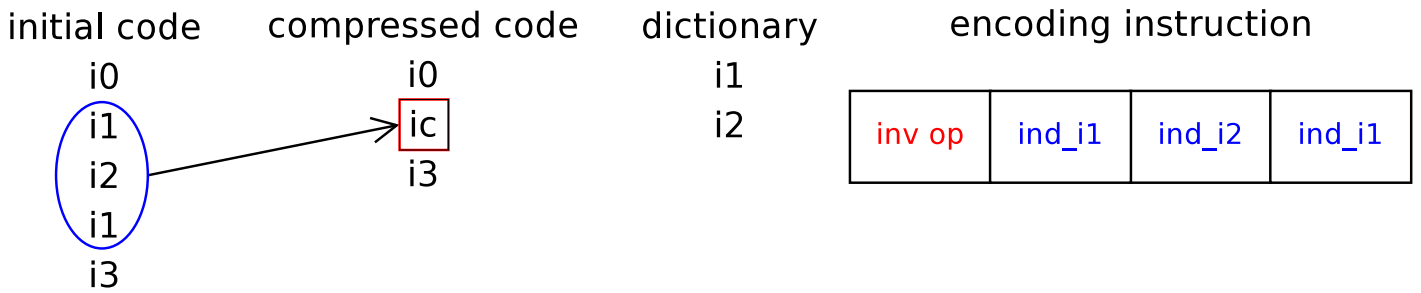
Principles of code compression

- The code size is reduced by compacting the code into a non executable format
- A decompression step is needed at runtime to retrieve the initial code

- Dictionary based compression is selected for the MORE project
  - Suffers from lower compression rate but allows simple and very fast decompression
- In-pipeline decompression is selected for the MORE project
  - The decompression engine is between fetch and decode stages
  - The cache contains compressed code and becomes virtually larger which results in fewer cache misses and fewer memory accesses

    $\Rightarrow$ Can improve performance and decrease energy consumption

Dictionary based code compression



- Build the dictionary with instructions from the initial code
- Compact these instructions into an encoding instruction containing corresponding dictionary indexes
- How to choose the instructions to include in the dictionary?
  - Most repeated instructions to improve compression rate
  - Most executed instructions to reduce instruction cache misses
  - Our solution: a mixture of both
    - P% of the dictionary is filled with the most executed and the rest is filled with the most repeated instructions
- Implementation of the transformation
  - Compressed instructions are annotated so in `OTAWA`
  - Simulator is modified to simulate the pseudo-compressed code

WCET related transformations

- The transformations considered to improve the WCET consist in linearizing the code (e.g. function inlining, loop unrolling)
  - Removal of flow control instructions improves predictability of processor states
- Implementation of transformations
  - Could be implemented by modifying the CFG in `OTAWA`
  - Most compilers already support this kind of transformations
  - A plugin is developed to control loop unrolling and function inlining through the `GCC Interactive Compilation Interface`

Current Section

Energy consumption evaluation

- A simple model of consumption based on the number and the cost of each kind of access to different memory types
- The cost of a read/write operation of each memory component is calculated through the `CACTI` tool
- Added probes to `OTAWA` simulator to obtain read/write access counts for each memory component (caches, DRAM, SPMs and dictionary)
- Finally, energy consumption is calculated by multiplying access costs by access counts

WCET analysis

- WCET analysis tool uses several components of `OTAWA`
  - instruction and data cache analyzers based on abstract interpretation techniques
  - a timing analyzer that evaluates WCET of basic blocks taking into account the target architecture and cache analysis
  - a WCET computation tool that builds an integer linear program according to IPET method which is then solved by the `lp_solve` tool
- Effects of code compression are taken into account by
  - considering the address of the instruction in the compressed code in instruction cache analysis
  - considering the effect of the decompression engine in basic block WCET evaluation
- The data cache analyzer interfaces with the data placement tool to get the information about the memory in which every piece of data is stored

Current Section

Test programs and hardware platform

- Experimented with 4 test programs
  - `adpcm`: adaptive pulse code modulation algorithm
  - `compress`: data compression program
  - `helico`: toy helicopter control program
  - `segmentation`: image segmentation algorithm
- Hardware platform
  - 2-way superscalar in-order processor
  - 1KB 2-way associative instruction and data caches - a small cache is chosen to get realistic results with the small test programs
  - To test data placement strategies
    - 512B 2-way associative data cache and 512B SPM

Impact of a data placement strategy

- Effects of *SPM_highFrequency* strategy

Benchmark	**Impact on energy**	**Impact on WCET**
`adpcm`	−13.3%	−23.8%
`compress`	−65.9%	−18.5%
`helico`	−7.7%	−5.9%
`segmentation`	−10.7%	N/A
average	−24.4%	−16.1

  - Traditional caches are much more energy hungry than SPMs and they are also very unpredictable which causes an over-estimation of WCET

⇒ Coupling a cache and an SPM with an adequate strategy can yield to important energy savings and improve WCET estimation compared to a single large cache

Global impact of code compression

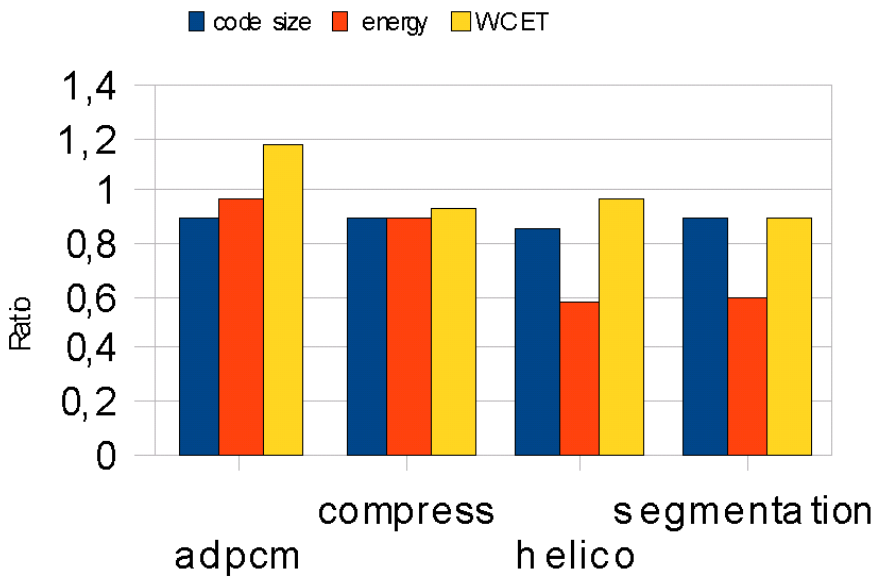
- P: the percentage of the dictionary filled with the most executed instructions
  - The lower the value of P the better the compression rate
  - The higher the value of P the better the execution time and the energy consumption reduction except where code compression creates conflicts misses
    - Since code placement with respect to the cache is modified by compression, instruction cache misses can increase in pathological cases
  - There is no direct correlation between P and the impact on the WCET
    - It can even be degraded!

$\Rightarrow$ A careful trade-off is necessary to meet system requirements

- Beware of pathological cases!

Effects of code compression

- Effects of code compression with a different value of P for each benchmark that leads to a good trade-off



⇒ Code compression can improve the WCET and energy consumption while reducing the code size but fine tuning of each application is required

Function inlining

- Function inlining replaces calls to functions with their bodies
  - Code becomes more linear
    - $\Rightarrow$ over-estimation of the WCET gets smaller
  - Performance tends to increase because
    - call/return instructions as well as prologue/epilogue code of functions are removed
    - function body can be optimized for the context of the caller
  - Has negative effects too
    - code size is increased
    - temporal locality of accesses to the instruction cache is decreased

Effects of function inlining

Benchmark	Impact on WCET	Impact on code size	Impact on energy
`adpcm`	−1.5%	+45.5%	−5.2%
`compress`	−6.5%	+44.5%	−0.9%
`helico`	−7.4%	+95.9%	−73.9%
average	−5.1%	+62.0%	−26.6%

- Gain on WCET is moderate but a larger cache would help in benefiting more from inlining
- Inlining significantly expands code size
  - Strategies to trade-off between the code size and the WCET should be set up
- Inlining decreases the number of instruction cache accesses but can also increase the number of misses

Current Section

Conclusion

- Embedded systems are subject to various constraints like code size, power requirements, execution time, etc.
- Code transformations can be necessary to meet these constraints, e.g.
  - code compression can improve code size
  - data placement strategies can improve energy consumption
  - limiting jump instructions can improve WCET estimation
- Experimenting several possible transformations is a costly and time consuming process

Conclusion and future work

- We have introduced a framework with the goal of hosting various transformations and measurements or analysis tools to facilitate the optimization process
- Using the framework, it is possible to
  - select the transformations that improve a target criterion
  - evaluate their effects on other important criteria
- The usability of the framework is shown with experimental results which suggest that
  - it is necessary to set up appropriate strategies to combine several transformations
- As a second part of the MORE project, we are developing an engine for iterative optimizations
  - controls the application of various transformations
  - tries to determine the best combination
  - takes into account system constraints

END

THANK YOU !!!

Questions?