

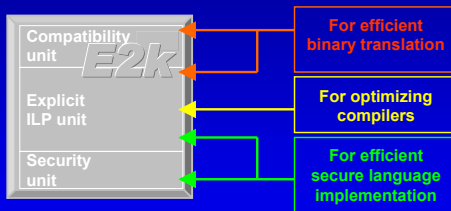
Optimizing Compilers for E2k Architecture

Vladimir Volkonskiy
Elbrus-MCST, Moscow, Russia
vol@mcst.ru

Agenda

- **Introduction**
- Optimizing compiler
- Optimizing binary translation system
- Secure language implementation
- Conclusions

Compiler designer's view on E2k microprocessor

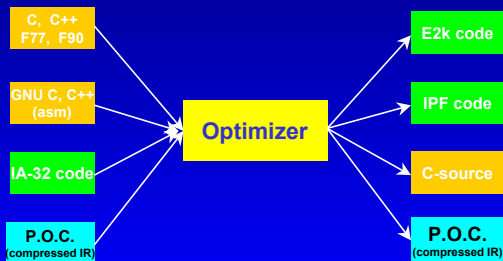


Compilers should find the best way for hardware units utilization

Agenda

- Introduction
- **Optimizing compiler**
- Optimizing binary translation system
- Secure language implementation
- Conclusions

Multiple Input & multiple Output Support

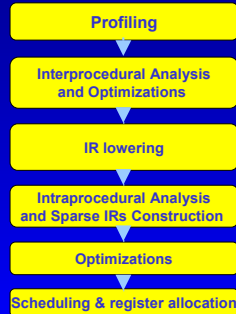


Semantic Modes

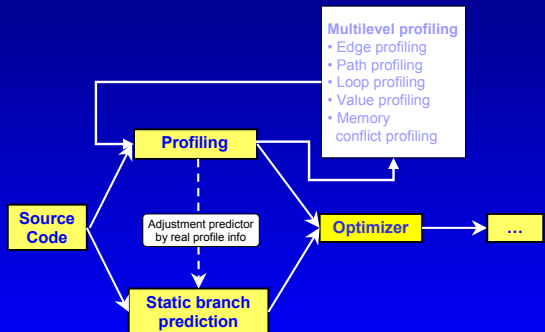
- **64-bit mode**: native mode for E2k architecture
 - Example: E2k Linux OS
- **32-bit mode**: used for compatibility reasons
 - Example: binary translation runtime support
- **Secure mode**: native with additional security features
 - Software in any standard-compliant programming language

Optimizer Major Components & Flow

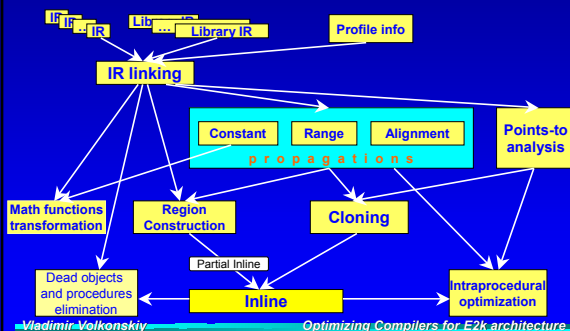
- Profiling and Static Branch Prediction
- Interprocedural analysis (IPA)
- Intraprocedural analysis
- Sparse Intermediate Representations
- Optimizations
- Scheduling & register allocation



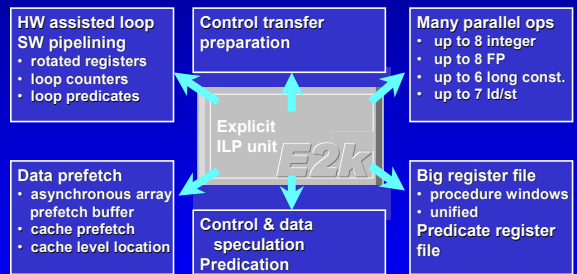
Profiling



IR Linking and Interprocedural Analysis & Optimizations



HW Support for Optimizations



Control Speculation Modes

Speculative mode can be applied to each operation

- branch to recovery code doesn't require special check operation
- the most universal mode

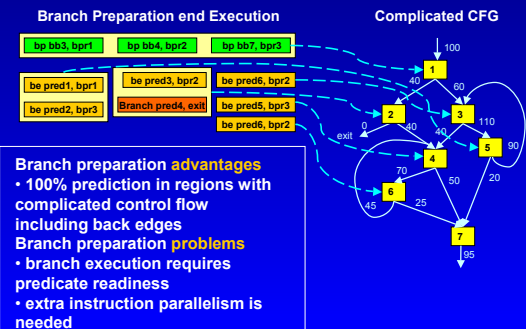
Semi-speculative mode can be applied to the most probable path

- no recovery code
- page miss exception can happen and is allowed
- special OS support is needed

Speculative mode without any losses

- excludes interlocks
- can be applied to the less probable path

Branch preparation



Asynchronous Array Prefetch

program trace with memory access interlocks (no optimization)

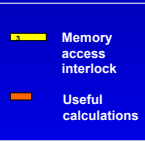
synchronous data prefetch in cache for next loop iteration

- there are some losses because of synchronous prefetch



asynchronous data prefetch in special Array Prefetch Buffer

- losses are equal to the longest memory access
- less cache pollution
- extra arithmetic units are freed for operations
- extra time is necessary for state forming and saving



Main Directions of Optimization

Control flow optimizations	→	Loop: unrolling, peeling, fusion, nesting, etc. Acyclic region construction
Data flow analysis and optimizations	→	Constant & Bits propagation, RLE, RSE, CSE, DCE, GPP, Value numbering, Strength reduction, Loop invariant removing
Hot region unloading	→	Loop: unswitching, vector invariant removing
Dependence analysis and optimizations	→	Loop nest static & dynamic analysis, Critical path optimization
Expose the power of EPIC/LIWI HW	→	If-conversion, Tail duplication, Loop SW pipelining, MAW & vectorization (MMX, SSE), Peephole, Predicate minimization, ...
Memory hierarchy optimizations	→	Loop: interchange, blocking; APB & cache prefetch, Data and code distribution, ...

Scheduling & Register Allocation

Acyclic region instruction scheduler

- for hyperblocks

Pipelined loop instruction scheduler

- for rather long running innermost loops
- rotating register & predicate allocation

Global instruction scheduler

- for regions with complex control flow
- integrated with E2k specific optimizations
- improves performance up to 20%

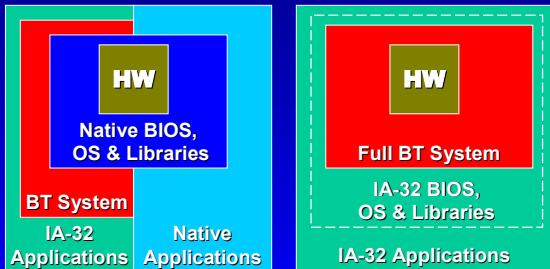
Global register allocation

- the best algorithms adapted to predication, register rotation and wide instruction parallelism

Agenda

- Introduction
- Optimizing compiler
- **Optimizing binary translation system**
- Secure language implementation
- Conclusions

BT for Applications vs. Full BT

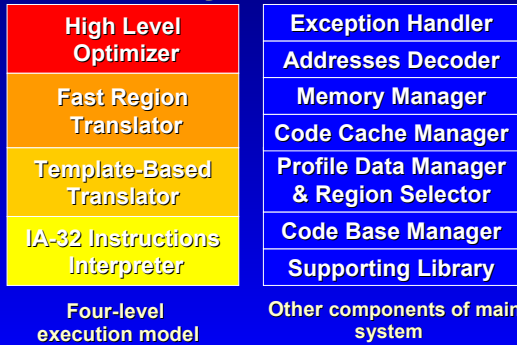


HW Support for Binary Translation

<ul style="list-style-type: none"> ✓ Two memory spaces ✓ Compatible TLB & MMU design, memory segments support, code & I/O pages protection ✓ Global IA-32 registers, rotating FP stack ✓ IA-32 flags calculation ✓ IA-32 compatible FP/MMX/SSE 	<ul style="list-style-type: none"> ✓ Recovery Points Support ✓ Specific memory disambiguation (Memory Lock Table) ✓ IA-32 to E2k Addresses Translation Cache (Table Cache) ✓ Asynchronous interrupts control ✓ Code Base Support ✓ Special mode for speculative FP/SSE execution
---	--

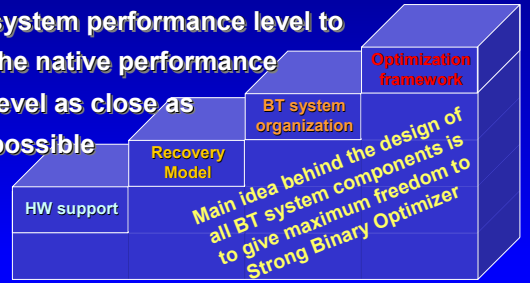
Support for IA-32 compatibility ← **Compatibility unit E2k** → Support for Binary Optimization

E2k BT System Overview

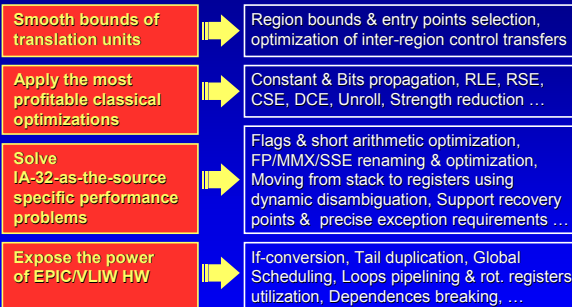


Strong Binary Optimizer

is responsible for raising the BT system performance level to the native performance level as close as possible



Main Goals of Binary Optimizer



Do it all as fast as possible !

Two E2k Optimizers

Strong Common Coding Standard

E2k Native Optimizing Compiler

E2k Binary Optimizing Compiler

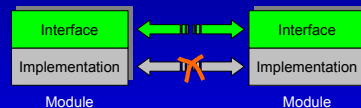
Level of Common Interfaces

IR, Data Flow & Control Flow Analysis results, Memory Manager, Supporting Tools, CG

Agenda

- Introduction
- Optimizing compiler
- Optimizing binary translation system
- **Secure language implementation**
- Conclusions

Concept of Secure Mode



- Several program modules run in common virtual space
- Set of entities comprising each module is divided into interface and implementation parts
- An error-free module that is managed through its interface should work properly
- Providing security means to ensure that any interaction between modules goes strictly through their interfaces

HW Support for Secure Language Implementation

Pointers integrity

- special tags
- alignment in memory



Memory cleaning

- register window
- user stack
- non-Initialized data
- Hidden chain stack

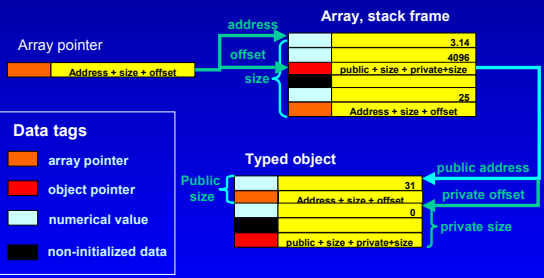
Correct operations on pointers

- transformations between different pointer types
- memory access by pointers
- procedure call

Runtime HW checks on operations with pointer

- object bound violation
- procedure entry points
- code & data correspondence at control transfer
- dangling pointers
- access rights

Tagged data



Programming Languages in Secure Mode

- **C** – full implementation of universal low-level programming language
 - Restriction: one can not access memory through a pointer obtained by conversion from integer
- **C++** - full implementation of complicated language for object-oriented & generic programming
 - Restriction: one cannot use placement `new` on a piece of memory allocated in unspecified manner

Bugs Detection in Secure Mode

- **Old programs:** SPECint 92 & 95: practically each task has a problem:
 - Array bounds violation: 27
 - Non-initialized data access: 4
 - Platform-dependent code usage: 189
- **Modern programs:** SPEC 2005 candidates:
 - 426.hydra: dangling pointer access
 - 444.namd: non-initialized data access
- **E2k software** is permanently verified

Agenda

- Introduction
- Optimizing compiler
- Optimizing binary translation
- Secure language implementation
- **Conclusions**

Main results

- **Optimizing compiler**
 - SPECint95 = 20, SPECfp95 = 56 for 300 MHz E2k (simulated)
 - compiler itself, OS Linux, user applications run correctly (several million lines)
 - retargeted to IPF and Elbrus-90 (SPARC compatible CPU)
- **Binary translation system**
 - faster than the best Xeon in terms of logical speed (simulated)
 - OS Linux & Windows run correctly (simulated)
 - Retargeted to IPF (high level optimizer)
- **Secure language implementation**
 - is used as verification tool for E2k software

Thank you!

Q & A