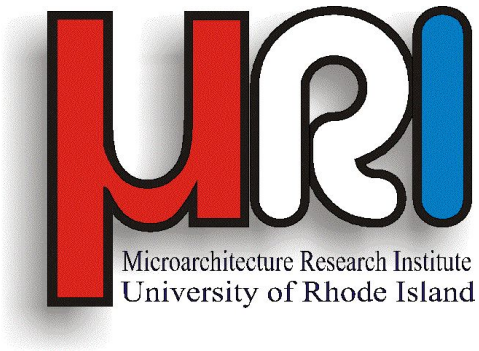
Characterizing Simple Single-sided Hammock Conditional Branches

Alireza Khalafi
David Morano
David Kaeli
Northeastern University

Augustus Uht
University of Rhode Island

Northeastern University
Computer Architecture Research



February 14, 2003

outline

- introduction
- examples
- detection
- experimental methodology
- characterization results
- summary

introduction

- **Hammock styled branches have some defined characteristics**
  - single entry
  - single exit
  - control-independent code at exit (join of the branch)
  - enclosed code blocks are generally small meaning that they fit inside the issue window of the processor
  - often have few data dependencies or a tractable number to deal with
- **offer an opportunity for ILP extraction**
  - the control-independent code following may be executed on the first detection of a the Hammock
  - for double-sided, both sides can be executed either simultaneously or better yet -- disjointly !
  - data dependencies are tracked in some way and only those control-independent instructions with those dependencies need be re-executed
- **how important are S-S Hammocks for IPC gains ??**

types of Hammock branches

- **types of Hammocks**
  - simple vs. nested
  - single sided vs. double sided
- **nested Hammocks**
  - since a Hammock (either single or doubled sided) forms a code block with a single entrance and exit => they can be nested
- **double sided**
  - represents the traditional "if-then-else" construct in higher-level languages
- **single sided**
  - represents just the "if-then" construct
  - easier to detect than any other types
- **we only focused on Simple Single-sided Hammocks since that is the primary focus of most everyone else !**

examples

single-sided

```
i10     a = x ;
i20     if (a) {
i30                 b = y ;
i40     }
```

```
i10     a = x
i20     if (! a) goto i40
i30     b = y
i40
```

double-sided

```
i10     a = x ;
i20     if (a) {
i30                 b = x ;
i40     } else {
i50                 b = y ;
i60     }
```

```
i10     a = x
i20     if (! a) goto i50
i30     b = x
i40     goto i60
i50     b = y
i60
```

another example

single-sided

```
i10     a = x ;
i20     if (a) {
i30                 b = x ;
i40                 c = z ;
i50     }
```

```
i10     a = x
i20     if (! a) goto i80
i30     b = y
i40     goto i70
i50
i60
i70     c = z
i80
```

- **I ignored these type of constructs in this experiment (not treated as an S-S Hammock)**

detection

- **very difficult to detect at run time alone**
- **so we use both compile time and run time compnents**
- **at compile time**
  - compile and link program files normally using existing tools
  - scan program object file looking for executable code sections
  - scan executable code for S-S Hammocks
  - put all found S-S Hammocks into a DB indexed by instruction address
  - write out the DB to a file (organized for fast pre-hashed access queries)
- **at run time**
  - simulate target program normally
  - when a conditional branch is encountered, lookup instruction address in the S-S Hammock DB file
  - if DB access shows a match, record or process as desired

detection algorithm (compile time)

- **read instructions as if we were executing the code**
- **if a conditional branch is encountered, record it and its target**
- **follow the not-taken path of the branch outcome**
- **if the branch target is reached before encountering either**
  - another conditional branch
  - an indirect jump
  - unconditional branch *
  - a subroutine call *
- **then we know that we have found a S-S Hammock**
  - there may be some that we missed
- **else, continue with the next conditional branch instruction and repeat**
- **reminder: any found S-S Hammocks are only "static" at this point**

* may still be a S-S Hammock in theory

detection algorithm (run time)

- **S-S Hammock DB file is mapped into memory for super fast access**
- **if we encounter a conditional branch :**
  - we access the hash table of the S-S Hammock DB (now entirely in memory)
  - we follow any hash chains until we either get a match or we get to the end of the chain
  - the whole matter of the S-S Hammock DB and access is encapsulated into a convenient object so that these details are invisible to the simulator programmer
- **found S-S Hammocks can now be recorded or processed as desired**

methodology

- **used 10 SpecInt programs**
  - 3 from SpecInt-95
    - GO, COMPRESS, IJPEG
  - 7 from SpecInt-2000
    - BZIP2, CRAFTY, GCC, GZIP, MCF, PARSER, VORTEX
- **compiled for MIPS-1 ISA using SGI native compiler ('-O')**
- **simulated execution for 600 M instructions**
- **data gathered after skipping the first 100 M**

results

of all instructions

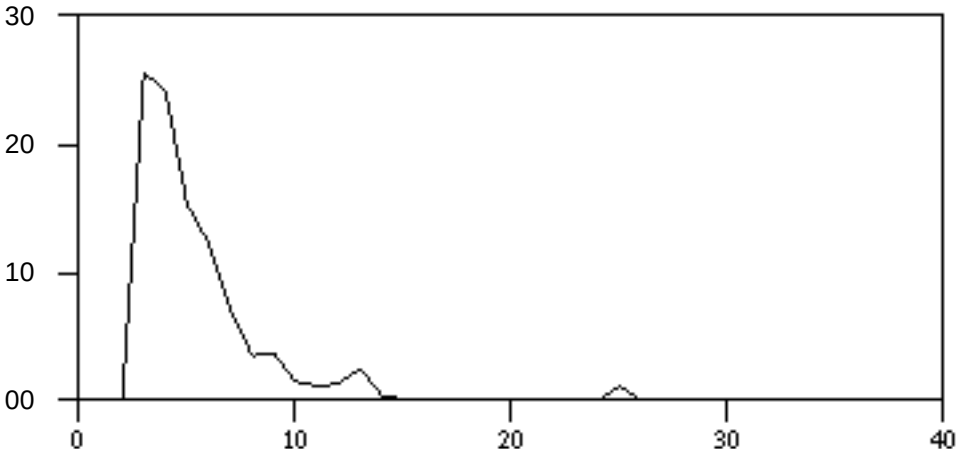
	% CF	% CB	% FWD CB	% SSH		% SSH of CB
bzip2	11.7%	9.1%	5.1%	0.2%		2.7%
compress	14.2%	9.4%	6.7%	1.7%		18.5%
crafty	10.8%	6.8%	6.3%	0.8%		12.3%
gcc	15.6%	12.4%	9.3%	0.6%		4.4%
go	11.9%	9.7%	7.9%	0.8%		8.8%
gzip	9.3%	6.7%	4.7%	0.6%		9.1%
ijpeg	6.0%	4.8%	2.9%	0.2%		3.7%
mcf	18.8%	14.6%	11.3%	0.7%		4.8%
parser	11.6%	7.6%	5.3%	0.5%		7.2%
vortex	12.2%	7.4%	6.0%	0.7%		10.7%
MEAN	12.2%	8.8%	6.5%	0.7%		8.2%

- CF **control flow**
- CB **conditional branches**
- FWD CB **forward c-branches**
- SSH **simple single-sided Hammocks**

extra results (SSH domain size)

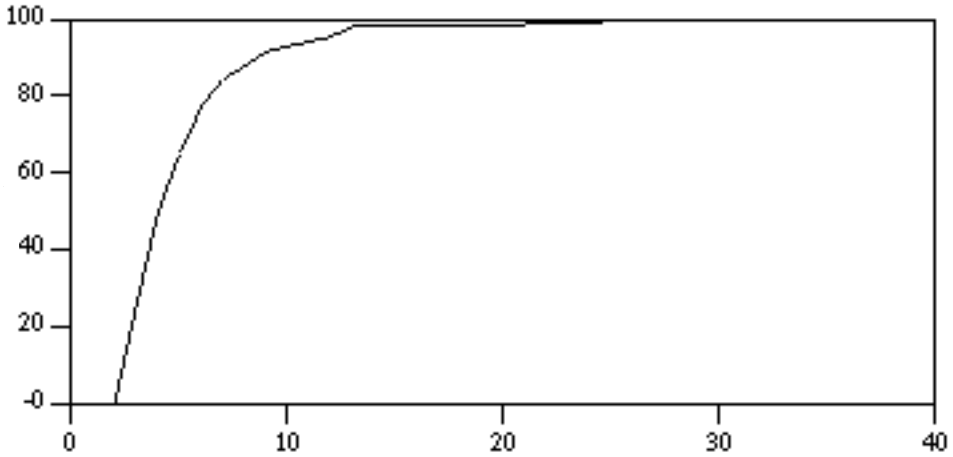
cumulative over all benchmarks

percent branches



density

percent branches



distribution

S-S Hammock branch domain size

summary

- **there are not very many S-S Hammocks in general integer code !**
  - this means that handling S-S Hammocks alone is probably not a big IPC win
  - so, more general ILP extractions techniques are needed and will do better than just handling S-S Hammocks alone
- **S-S Hammocks have relatively short branch domain sizes**
  - this is good for capturing the domains inside the instruction window !
- **more elaborate branch constructs could still be characterized and investigated**