Treegion Instruction Scheduling in GCC

Chad Rosier

GCC Developers' Summit

June 30th, 2006

TINKER Microarchitecture and Compiler Research Group
Department of Electrical & Computer Engineering
North Carolina State University

Presentation Outline

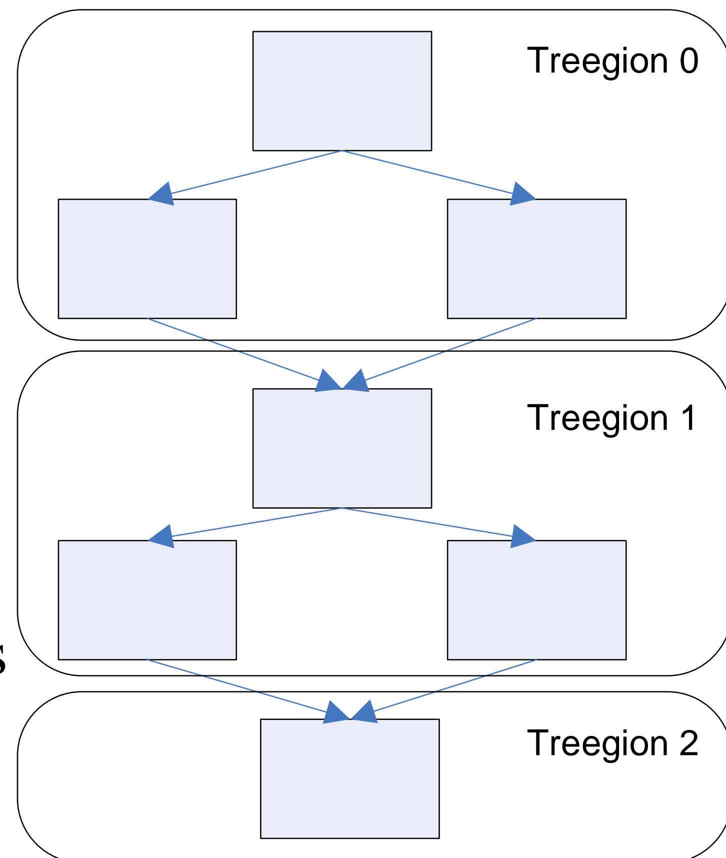
- Introduction
- Treegion formation
- Code size efficiency
- Implementation
- Summary
- Future work

Region Formation Techniques

- Trace Scheduling [Fisher and Ellis]
  - Linear regions, based on profile, optimizes likely path
- Superblock Scheduling [Chang, et al.]
  - Apply tail duplication to remove side entrances
  - Robert Kidd working at Tree SSA level
- Hyperblock Scheduling [Mahlke et al.]
  - Add hardware predication
- Regions are linear in nature

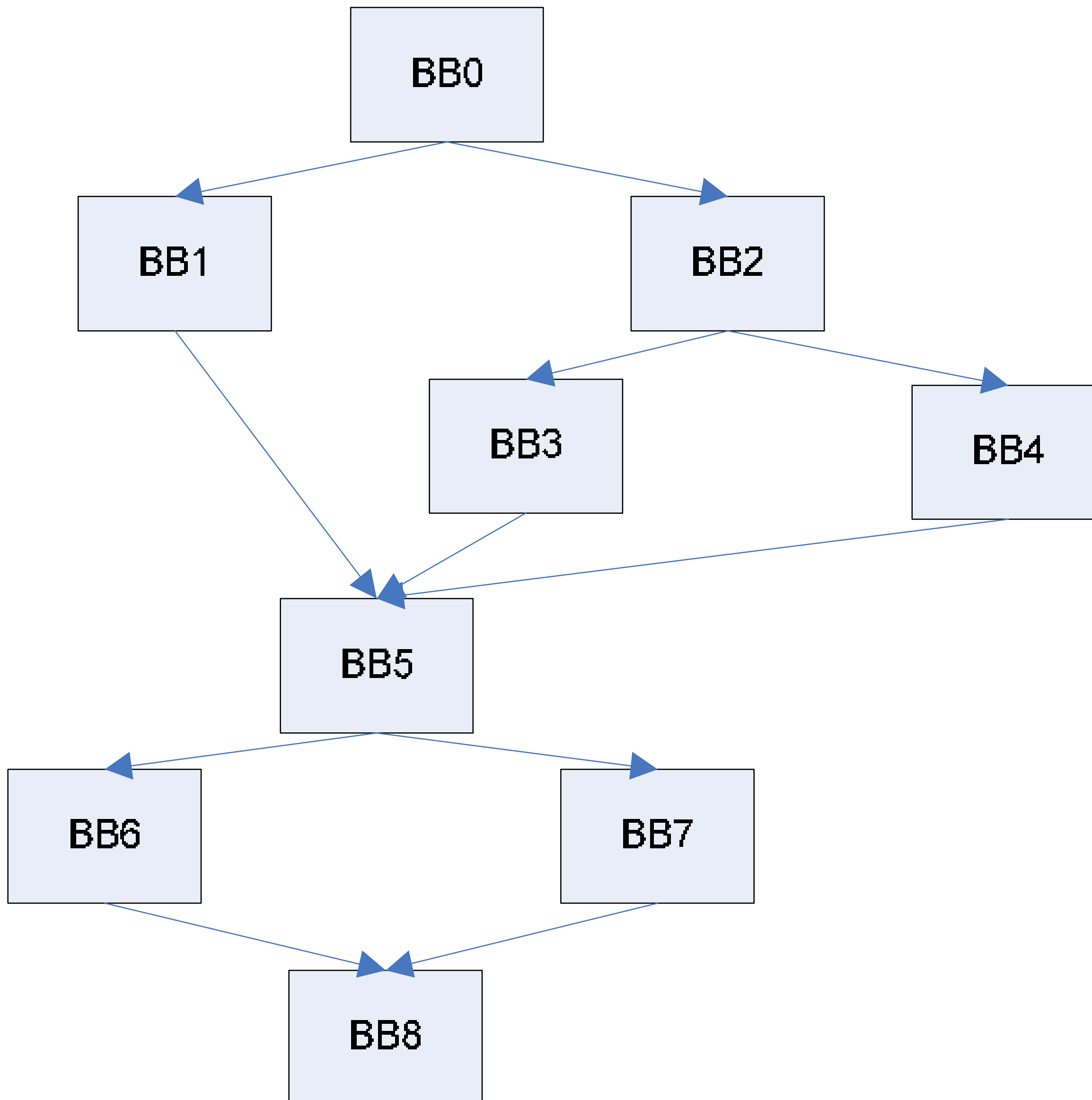
Treegion Scheduling [Havanki et al.]

- Tree-shaped subgraph of CFG
  - single-entry, multiple-exit
  - acyclic, non-linear
- Multiple paths considered for speculating instructions
- Formation based on CFG
- Profile used for scheduling
- Additional notable characteristics
  - Calculating domination
  - No merge points



Treegion Scheduling

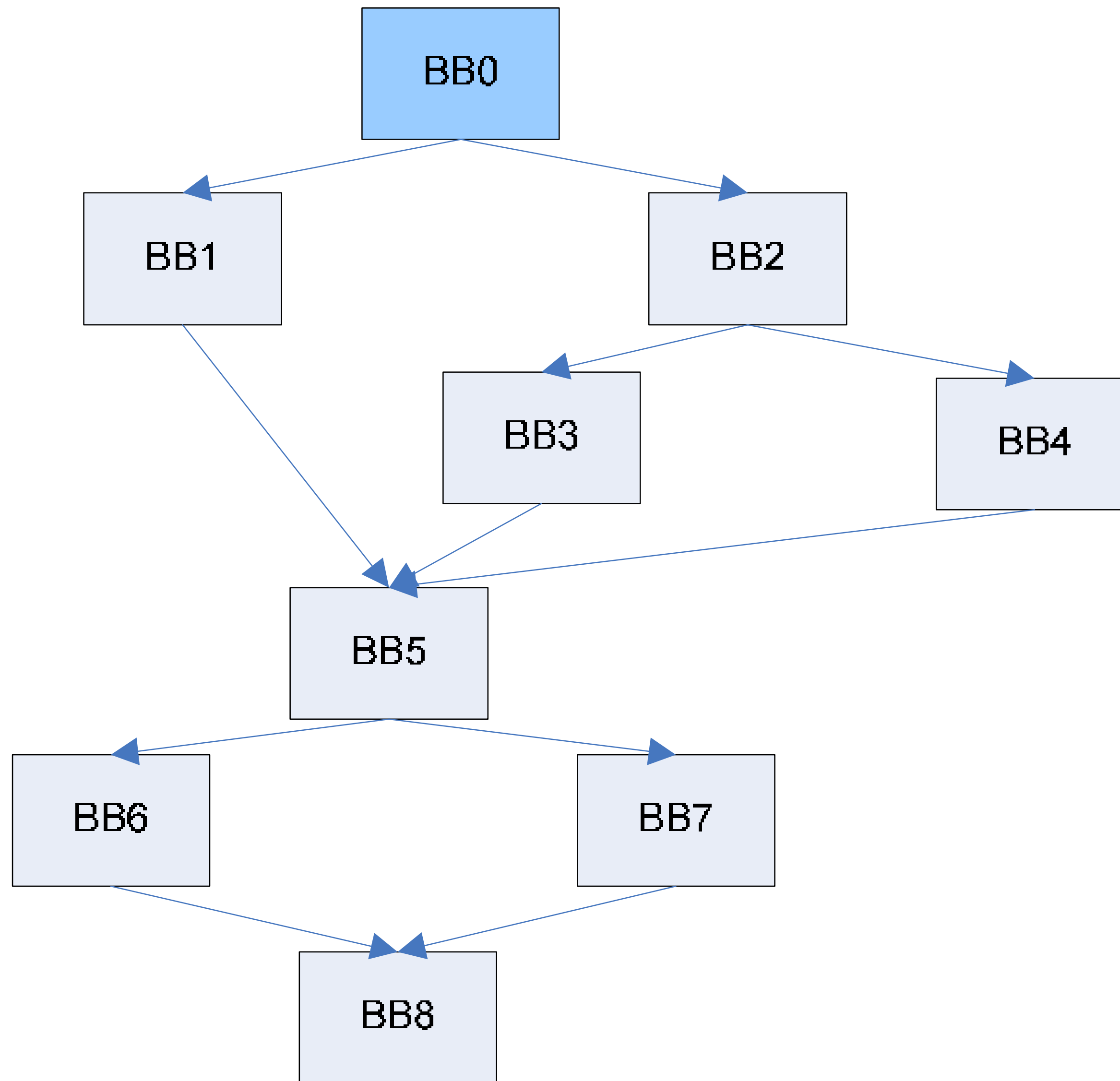
- Treegion formation (basis of this talk)
  - Natural Treegion formation
    - Treegion formation without tail duplication
  - Region enlargement optimizations
    - Tail duplication
- Treegion-based instruction scheduling
    - Currently using existing Haifa scheduler



Treegion: None

Successors: None

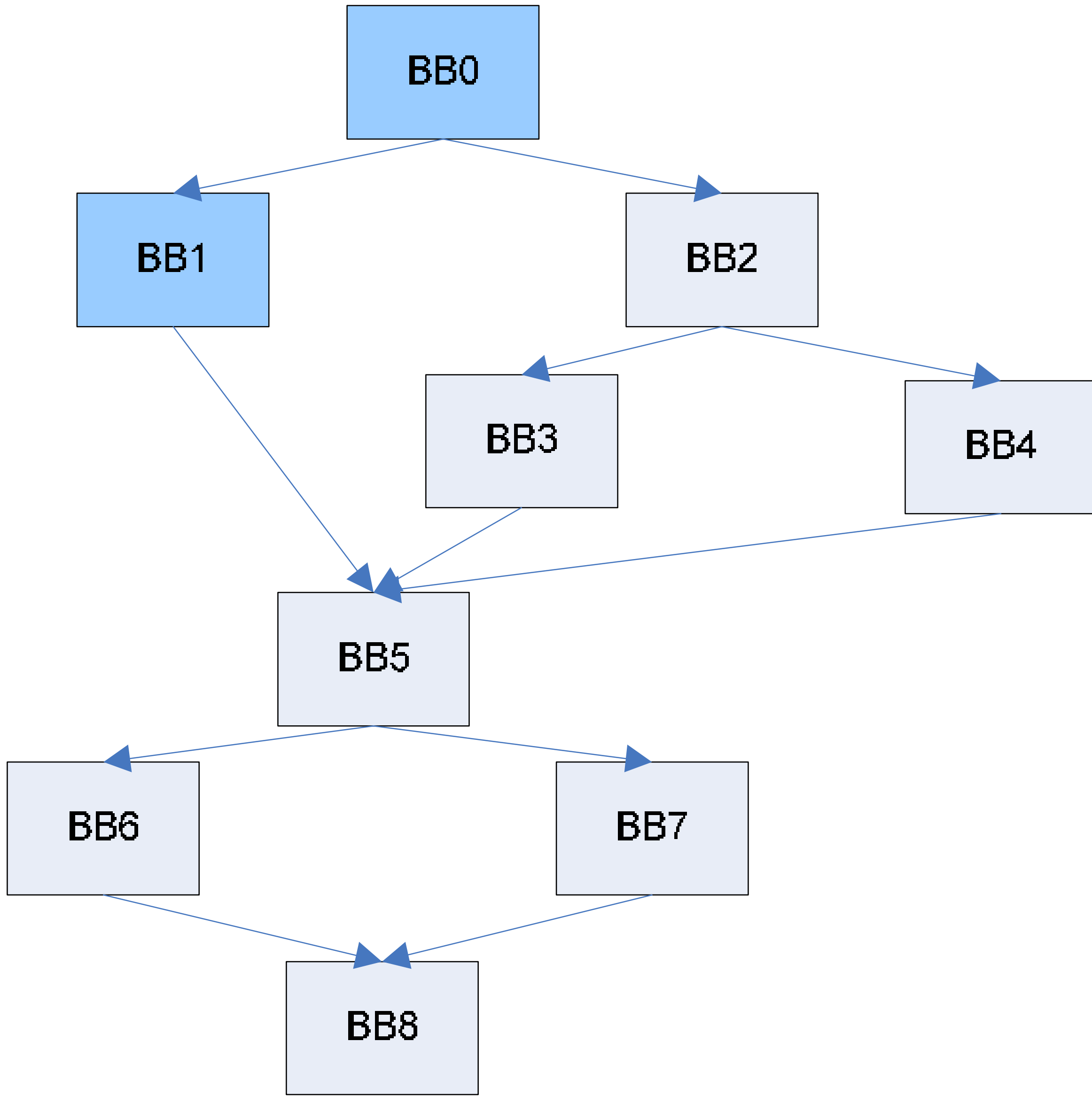
Saplings: None



Treegion: {BB0}

Successors: {BB1, BB2}

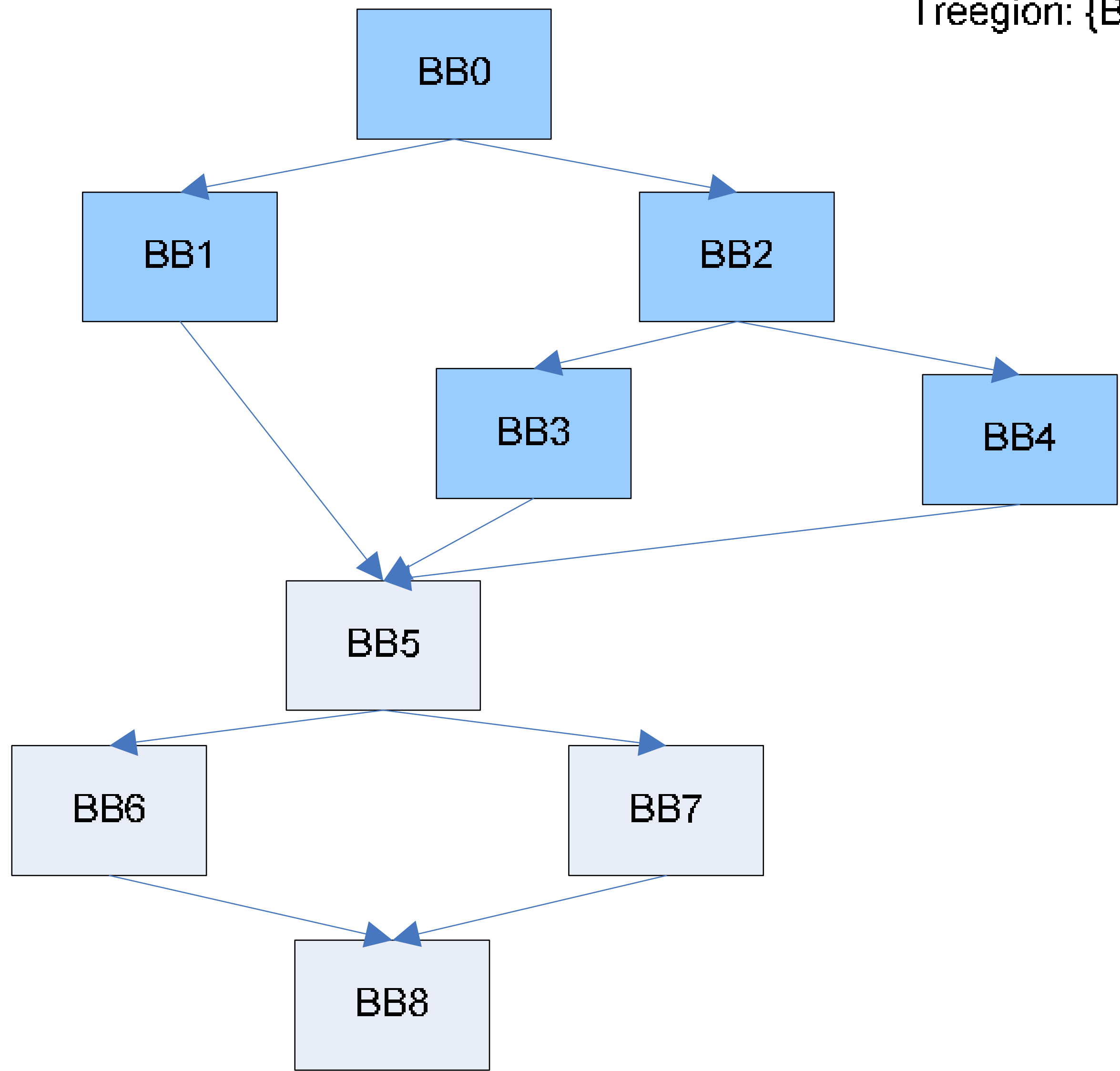
Saplings: None



Treegion: {BB0, BB1}

Successors: {BB2}

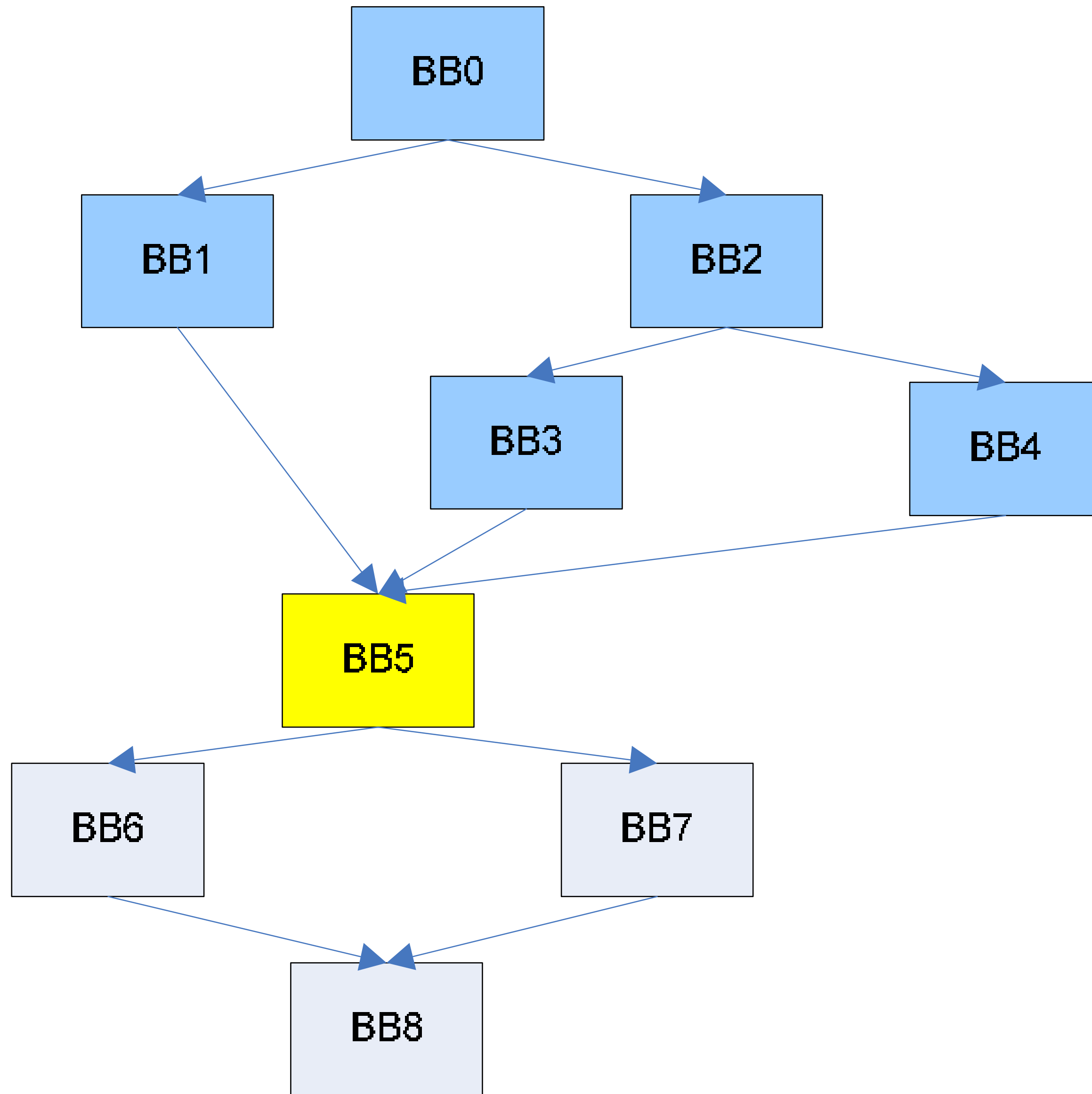
Saplings: {BB5}



Treegion: {BB0, BB1, BB2, BB3, BB4}

Successors: {}

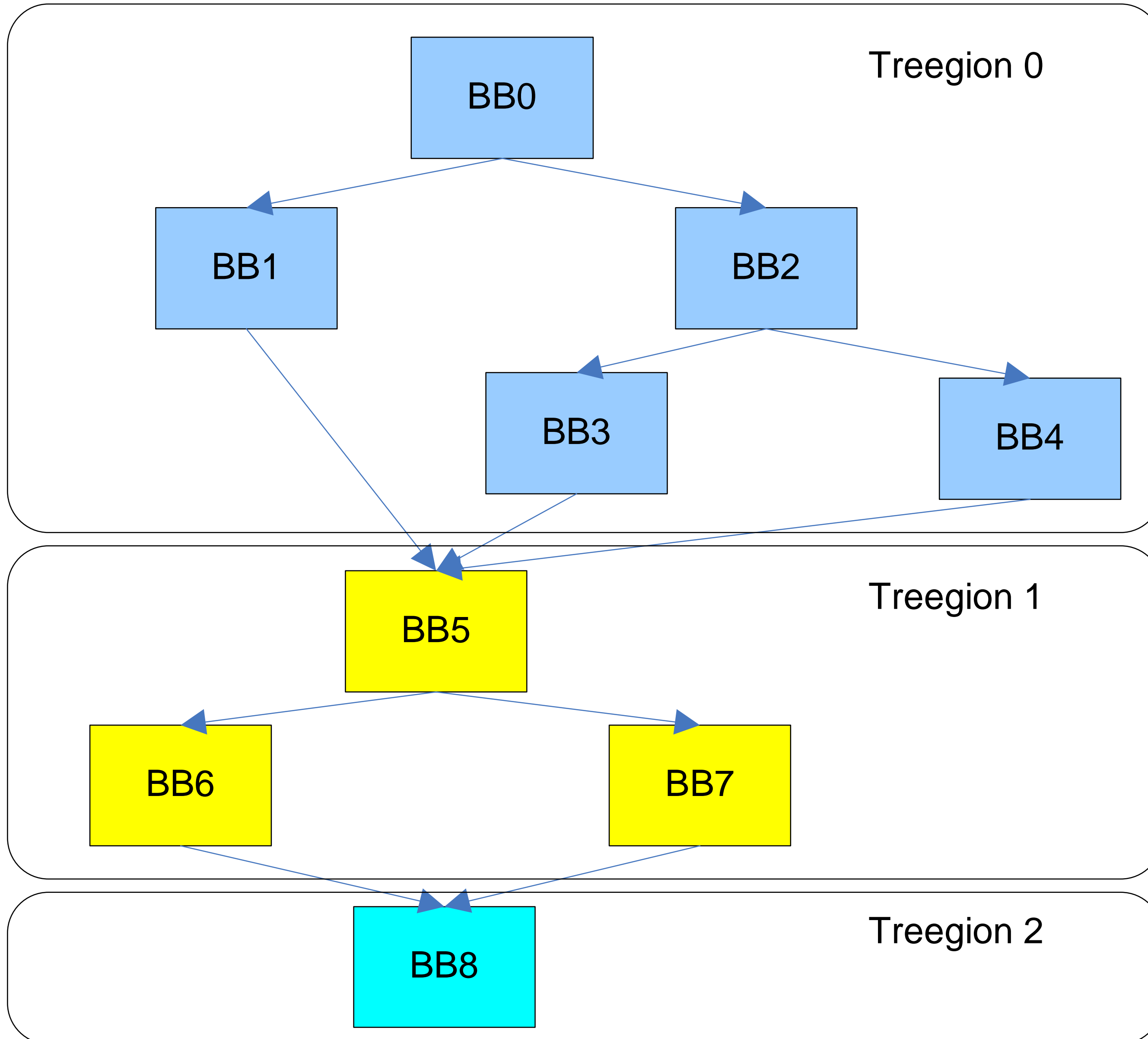
Saplings: {BB5}



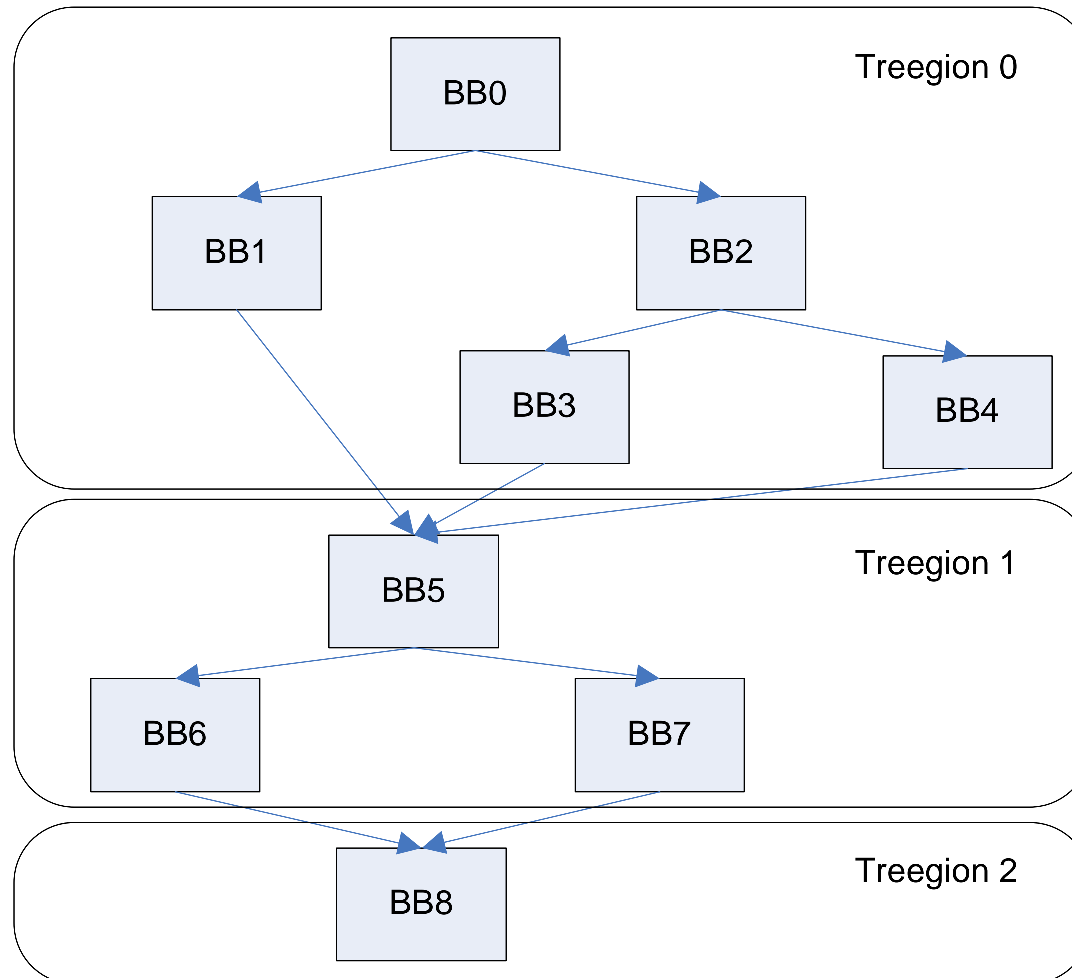
Treegion: {BB5}

Successors: {BB6, BB7}

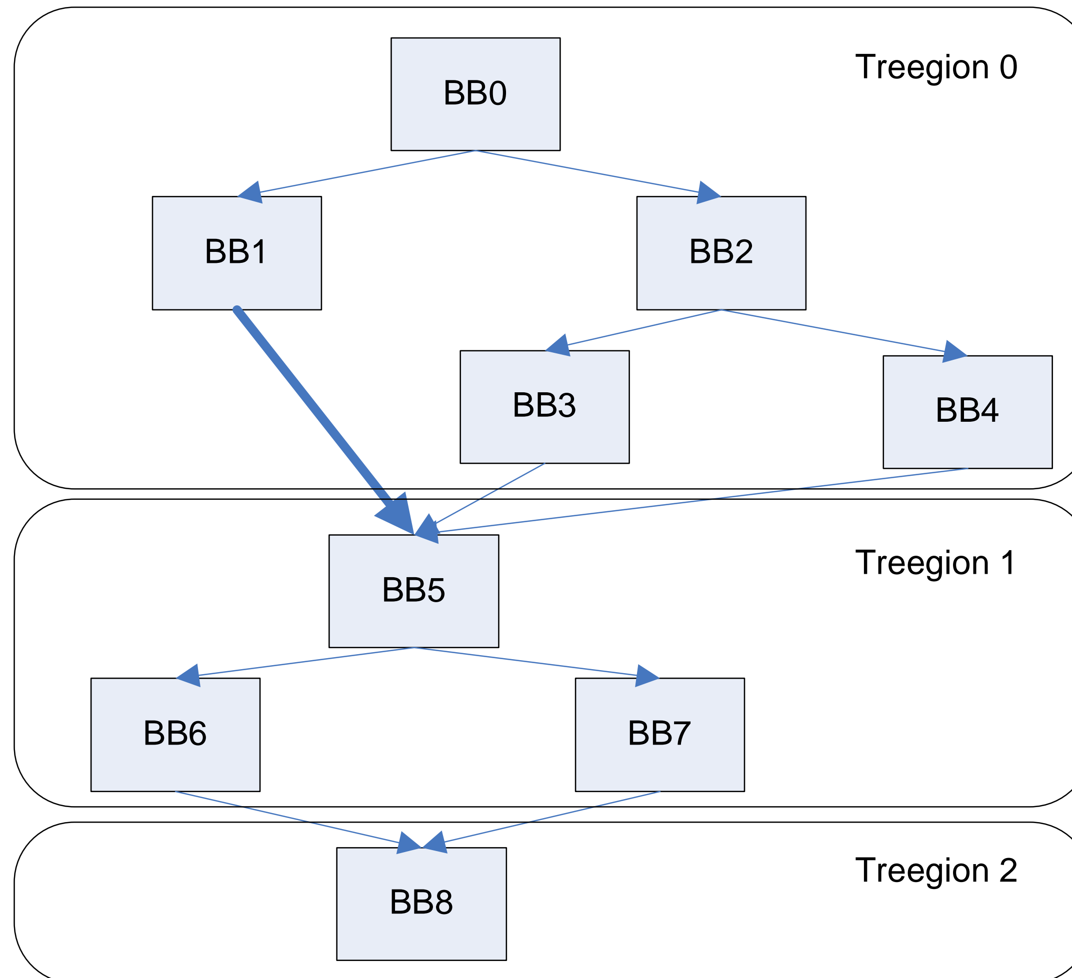
Saplings: None


Treegion 0
BB0
BB1
BB2
BB3
BB4
Treegion 1
BB5
BB6
BB7
Treegion 2
BB8

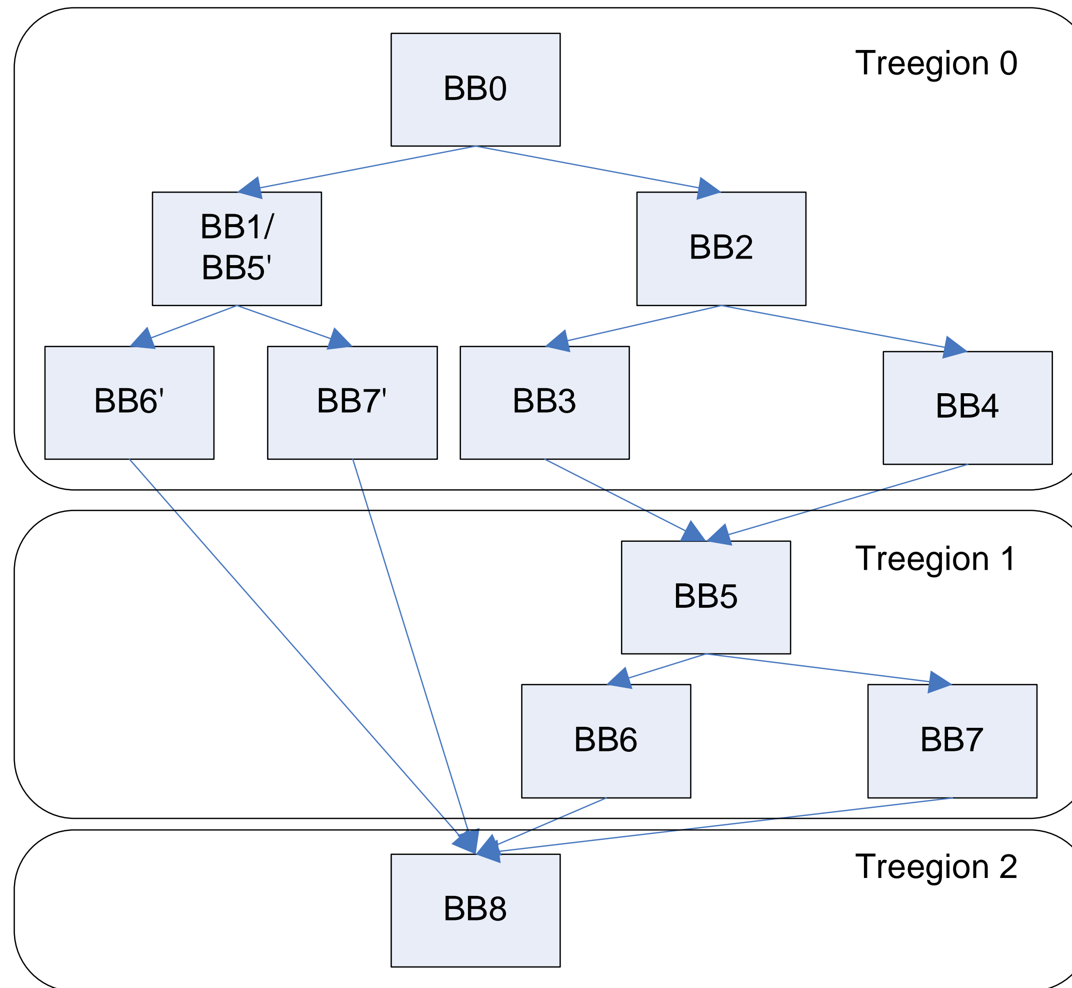
Tail Duplication Example



Tail Duplication Example



Tail Duplication Example



Region Enlargement Optimizations

- Instruction level parallelism (ILP) vs. static code size
  - Region enlarging optimizations usually enhance ILP
    - Cyclic scheduling: loop unrolling, loop peeling, etc.
    - Acyclic scheduling: tail duplication, recovery code, etc.
- I-cache and ITLB performance vs. static code size
  - Larger code usually means larger I-Cache footprint
- Trade off of the conflicting effects of code size increase
  - Especially in acyclic global scheduling

Code Size Efficiency

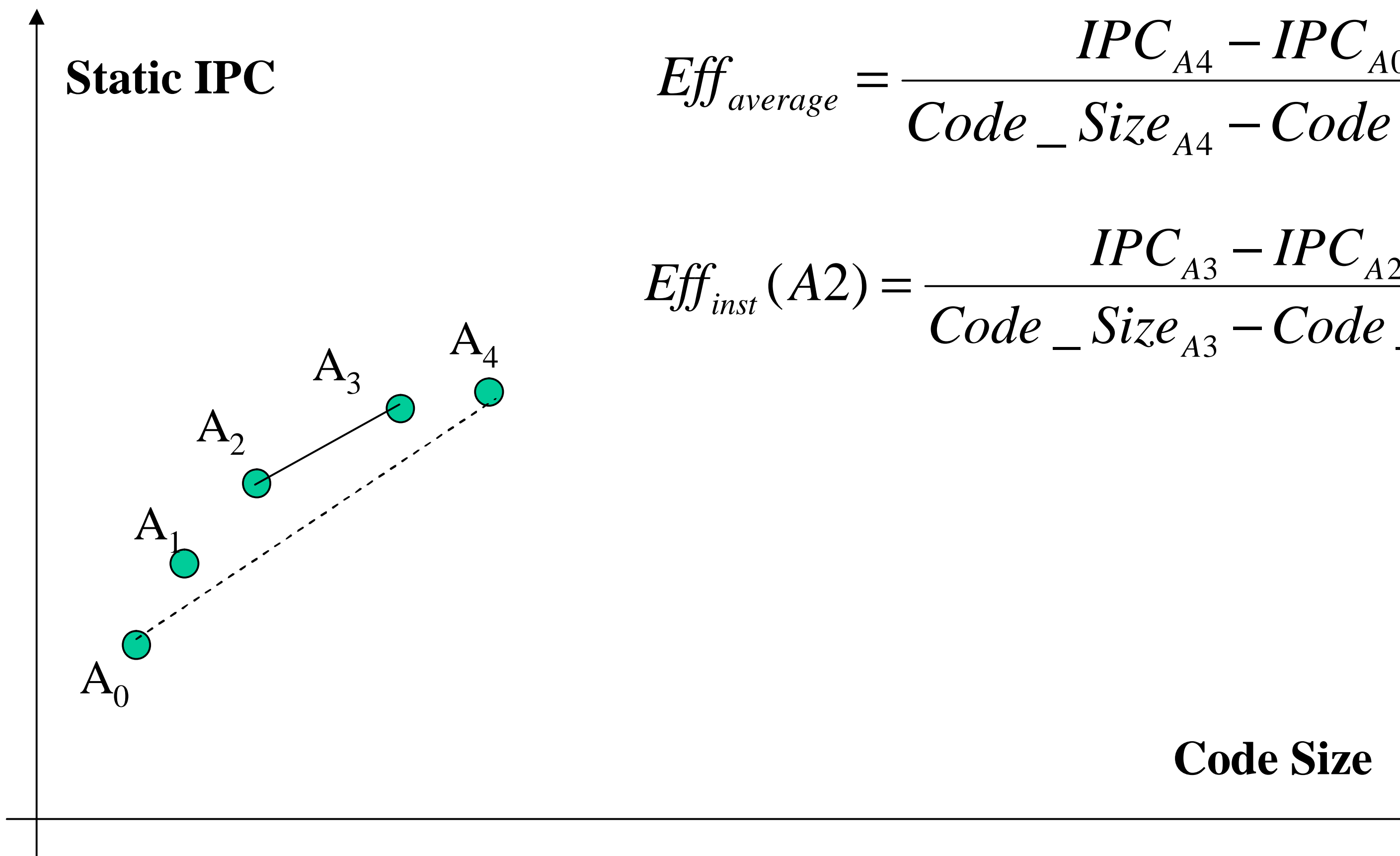
- Use the ratio of IPC changes over code size changes as an indication of code size efficiency.
  - Average code size efficiency

$$Efficiency_{average} = \frac{IPC_{candidate} - IPC_{natural_treegion}}{code_size_{candidate} - code_size_{natural_treegion}}$$

  - Instantaneous code size efficiency

$$Efficiency_{inst} = \frac{IPC_{after_individual_application} - IPC_{before_individual_application}}{code_size_{after_individual_application} - code_size_{before_individual_application}}$$

Instantaneous Code Size Efficiency



$$Eff_{average} = \frac{IPC_{A4} - IPC_{A0}}{Code_Size_{A4} - Code_Size_{A0}}$$

$$Eff_{inst}(A2) = \frac{IPC_{A3} - IPC_{A2}}{Code_Size_{A3} - Code_Size_{A2}}$$

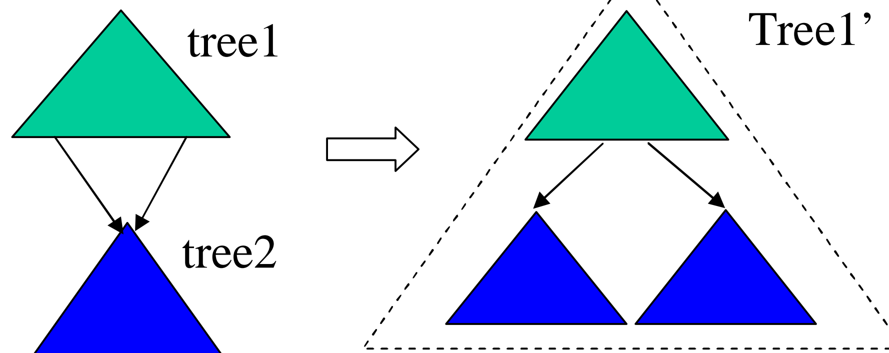
Estimate Static IPC Before Scheduling

- Use the expected execution time to calculate the static IPC
  - For a multi-path region:

$$Exe_Time_{Expected} = \sum_{path_i} \left[Max\left(data_dependence_bound_{path_i}, resource_bound_{path_i} \right) * Freq_{path_i} \right]$$

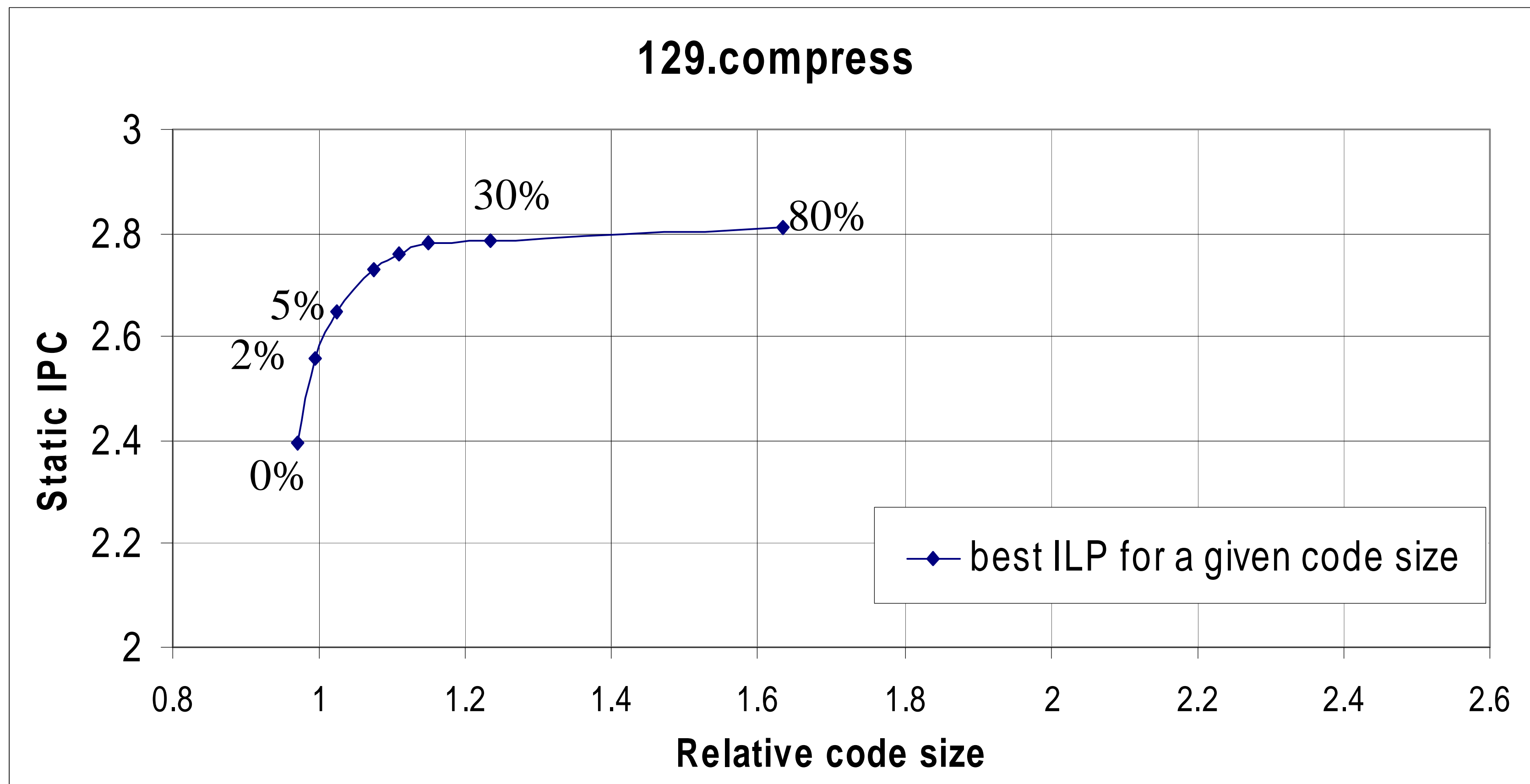
- Now, IPC changes can be calculated as execution time saved by the optimization.

Example:

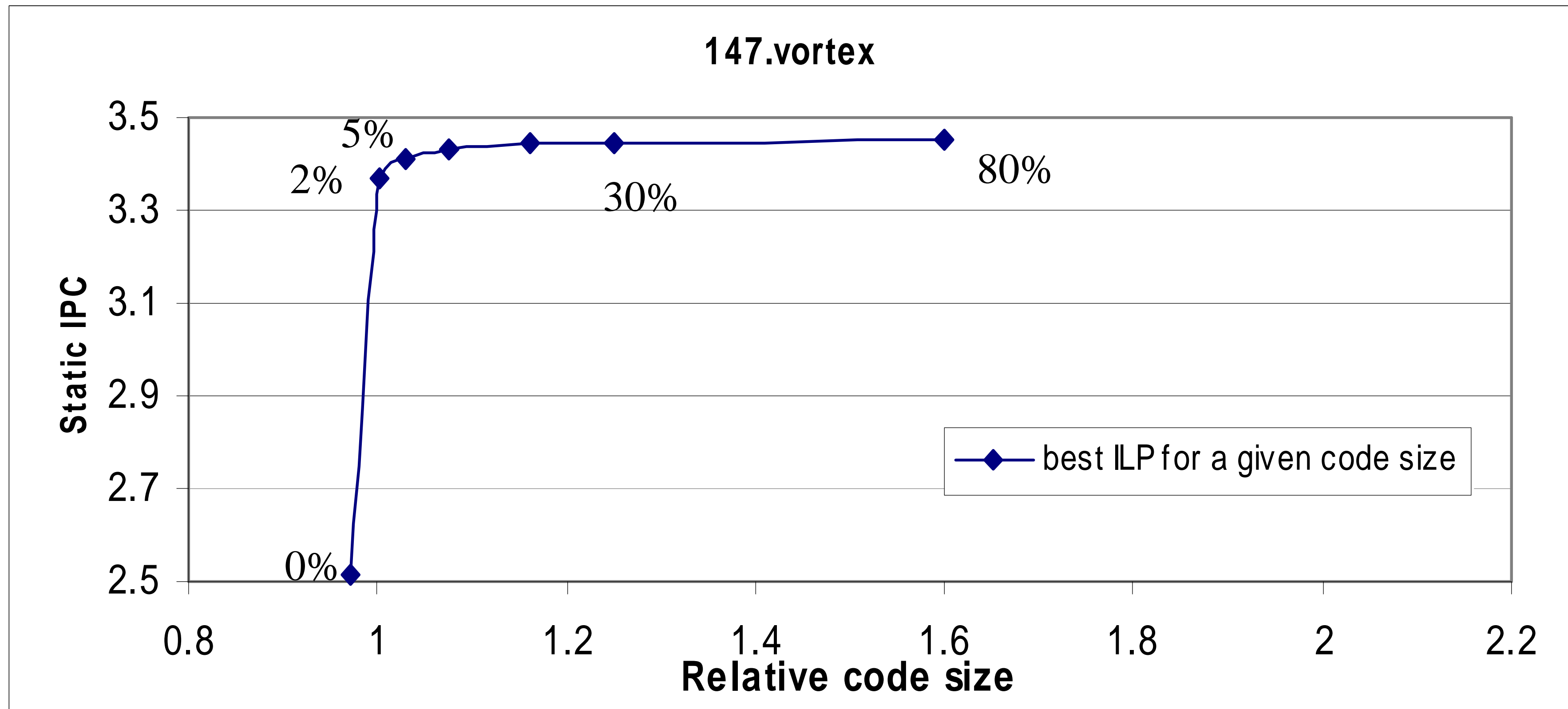


$$Eff_{inst}(A) = \frac{Exe_time(tree1) + Exe_time(tree2) - Exe_time(Tree1')}{Code_size(tree2)}$$

Motivating Results from LEGO



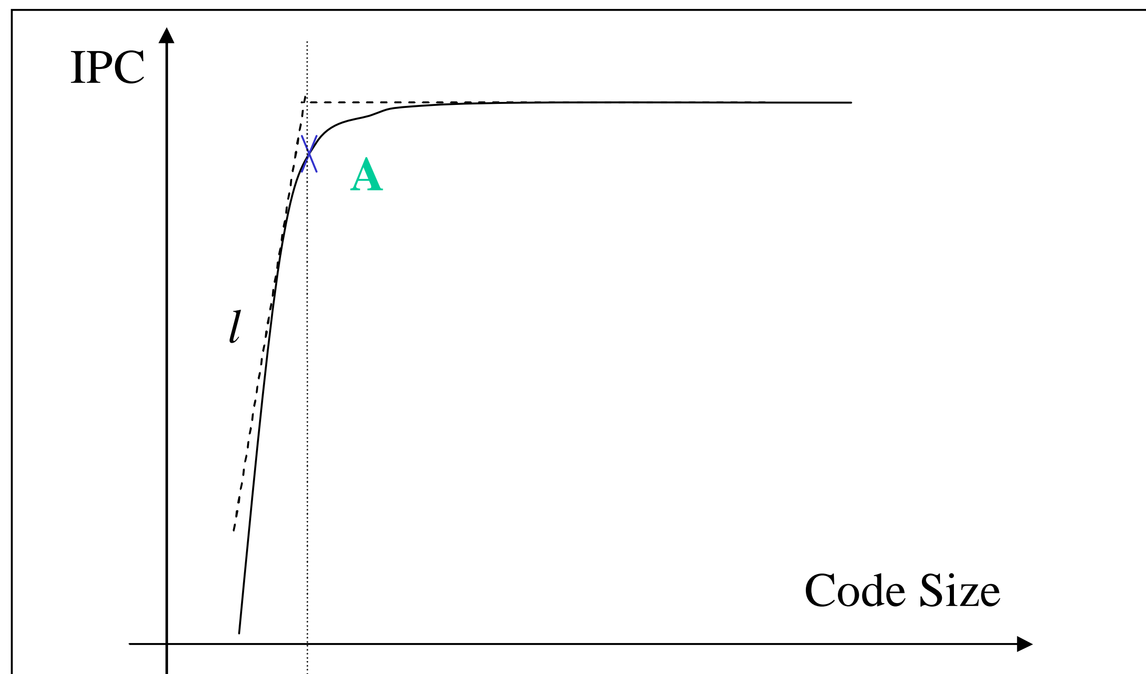
Motivating Results from LEGO



Reason: only a very small part of the program is frequently executed.

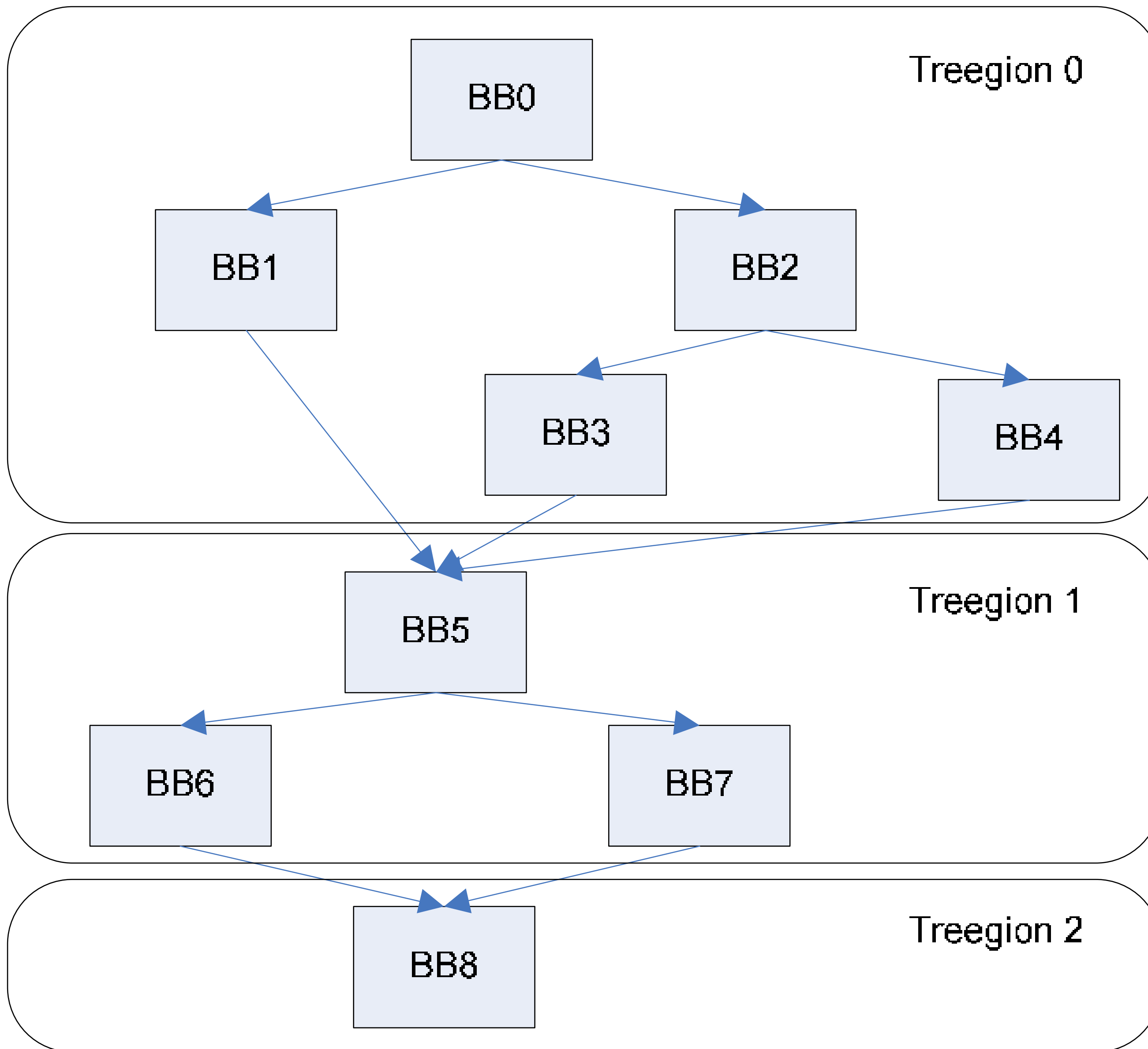
Optimal Code Size Efficiency

- Point where the ‘diminishing returns’ start
- Range between tan(π/12) (.268) and tan(π /6) (.577)

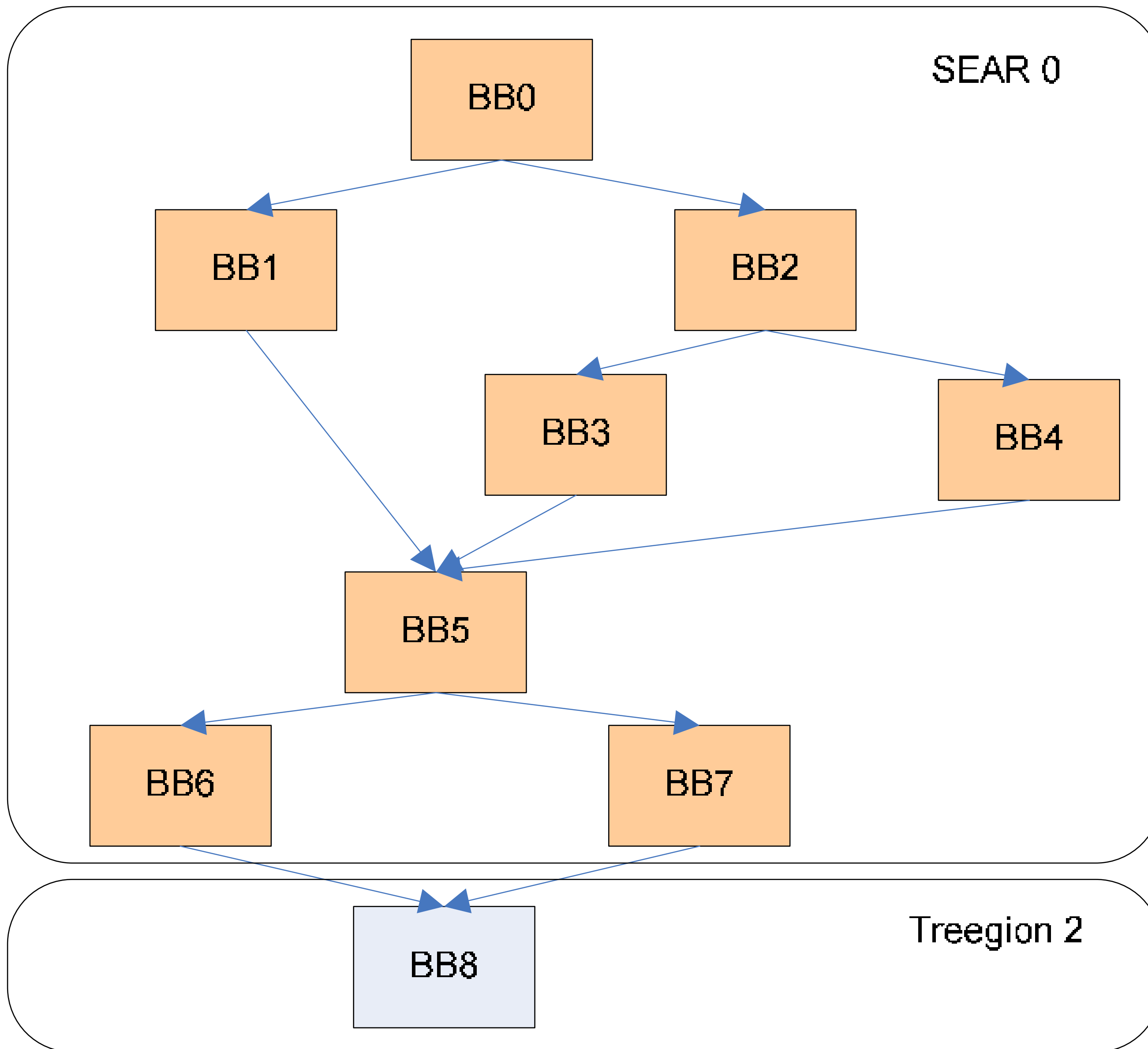


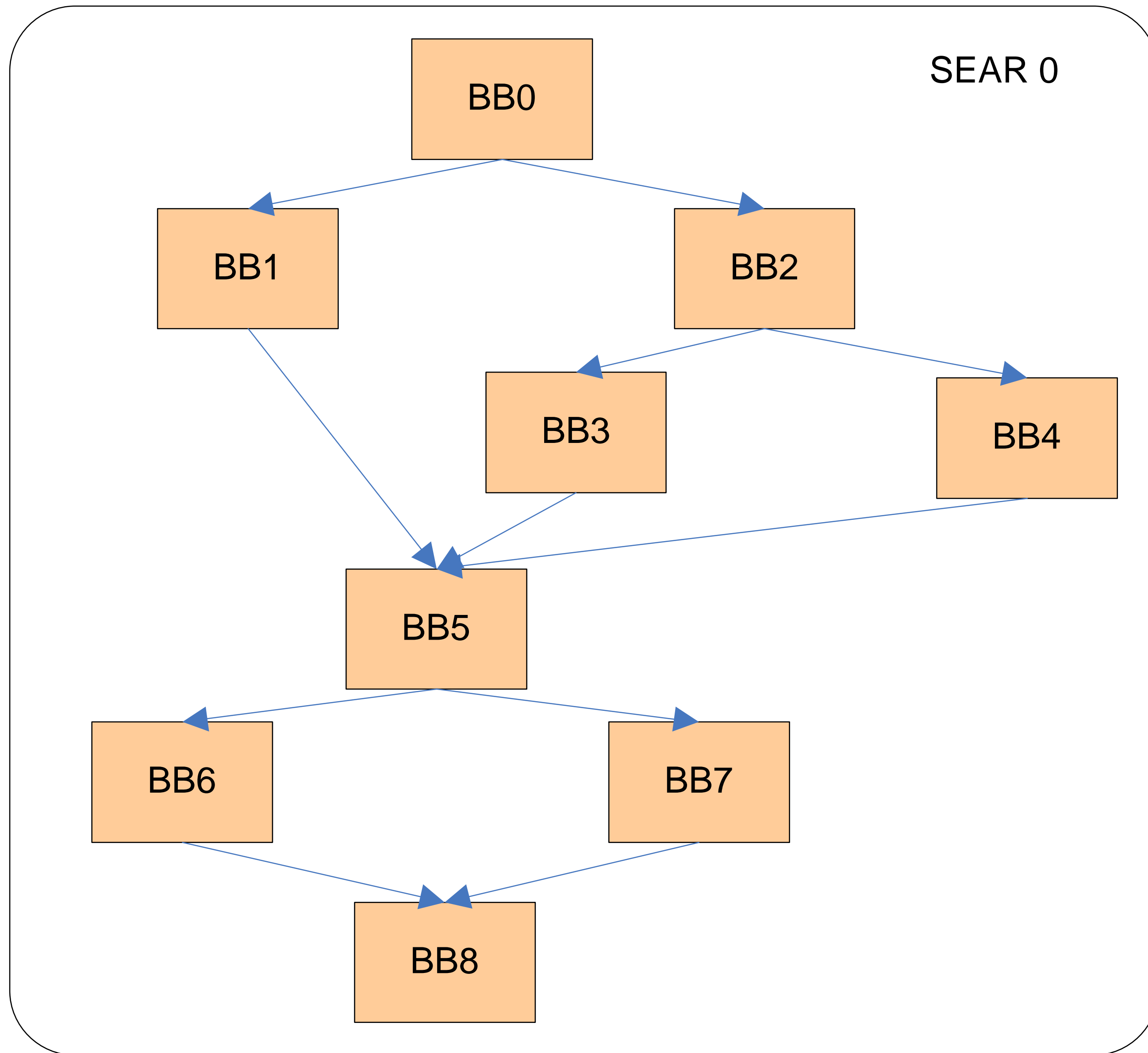
Single Entry Acyclic Regions (SEARs)

- Prompted by recent extend region code
- Form larger regions by merging treegions
  - IFF predecessor edges entering the child treegion are exit edges from parent region
    - Prevents side entrances (and need for compensation code)
- We now have merge points in the region so it is no longer a Treegion
- SEARs formation happens after tail duplication, but ICSE heuristic still applicable



Treegion 0
BB0
BB1
BB2
BB3
BB4
Treegion 1
BB5
BB6
BB7
Treegion 2
BB8


SEAR 0
BB0
BB1
BB2
BB3
BB4
BB5
BB6
BB7
BB8
Treegion 2

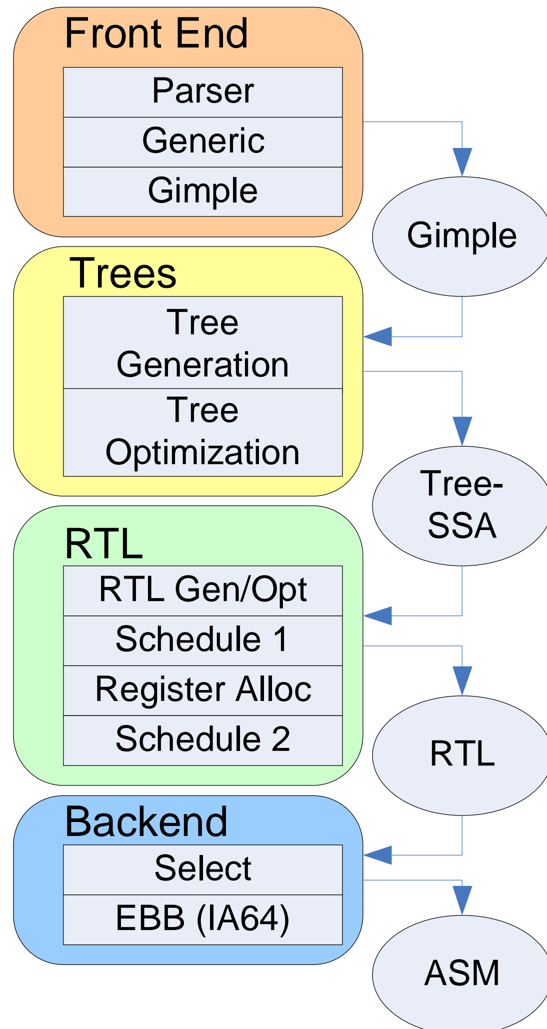

SEAR 0
BB0
BB1
BB2
BB3
BB4
BB5
BB6
BB7
BB8

Tree Traversal Scheduling

- Final step is to (Haifa) schedule instructions
- Other region formation techniques optimize likely path while off path suffers
- Prioritize likely path (based on exec frequency)

- TTS Algorithm:
  1. Sort basic blocks in depth-first traversal order
  2. List schedule beginning at root block
  3. Consider speculation of instructions dominated by current block
  4. Repeat step 3 until all blocks scheduled

Implementation Details



- Serves as front end for Haifa scheduler
  - Scheduling pass prior to RA
- Primary modifications in `sched-rgn.c`
- ICSE - data dependence code in `sched-dep.c`
- Tail duplication – duplication code available in `cfglayout.c`

Compile Time Considerations

- Efficient updating of region structures
- Calculating ICSE is expensive
  - Requires building DDG to find dependence bound
- Compile time considerations:
  - Limit duplication based on:
    - Number of blocks/instructions
    - Code expansion
    - ICSE threshold

Region Formation Statistics

	Region	Natural Treegion	Treegion (k = 0.577)	Treegion (100 insns)
# Basic Blocks	1.10	2.65	2.79	5.70
Instructions	8.66	20.89	21.70	35.95
Interblock	0.09	3.65	3.81	6.00

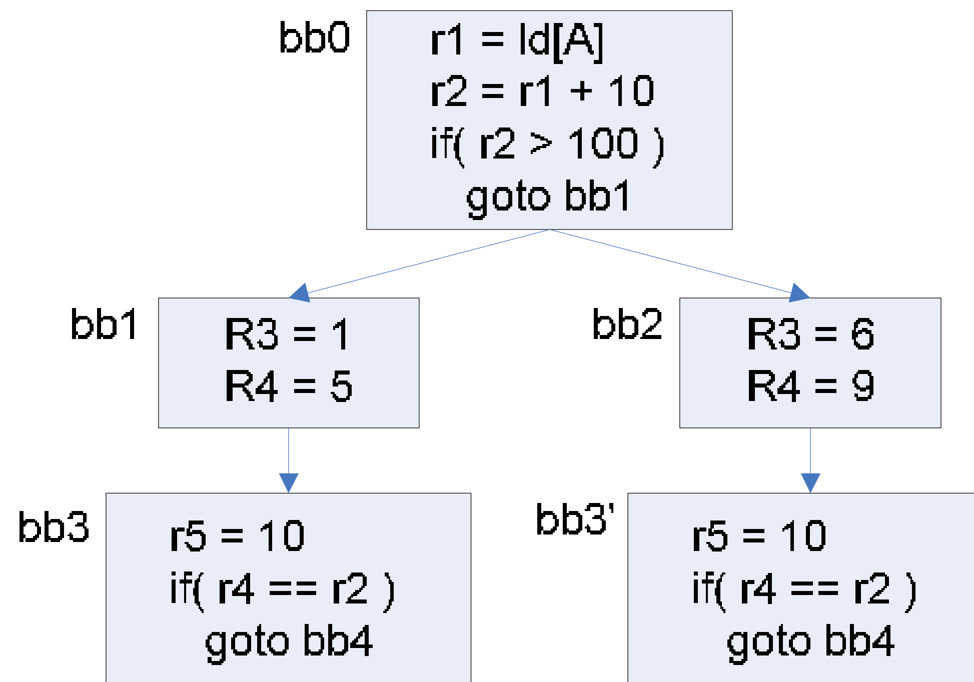
- Significant increase in region size
- Increase % blocks contained within multi-block region
- Limited additional duplication using ICSE
- Without code size consideration region size may become extremely large

Speedup

	Region	Natural Treegion	Treegion (k = 0.577)	Treegion (100 insns)
gzip	1.00	0.96	0.96	1.03
mcf	1.00	1.00	1.00	1.00
crafty	1.00	0.99	1.00	1.00
parser	1.01	1.01	1.01	1.01
gap	0.99	1.01	1.00	1.00
bzip2	0.99	1.06	1.06	1.06
twolf	1.03	1.01	1.01	1.03
wupwise	1.00	0.99	1.02	1.01
swim	1.00	1.02	1.04	1.01
mgrid	1.00	0.99	0.99	1.00
applu	1.00	1.00	1.00	1.00
equake	1.00	1.00	1.01	1.00
ammp	1.01	1.01	1.00	1.00
sixtrack	1.00	0.98	0.98	0.98
apsi	1.00	1.01	1.01	1.01
average	1.00	1.00	1.01	1.01

Operation Combining

- Tail duplication increases code size
- General operation combining reduces code size

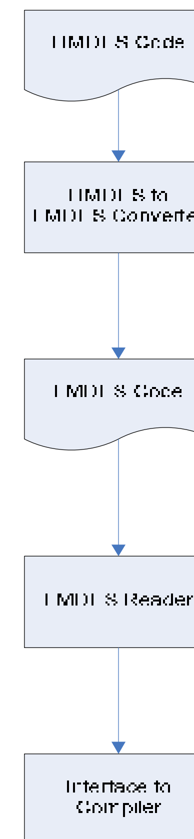


Automata Based Pipeline Description

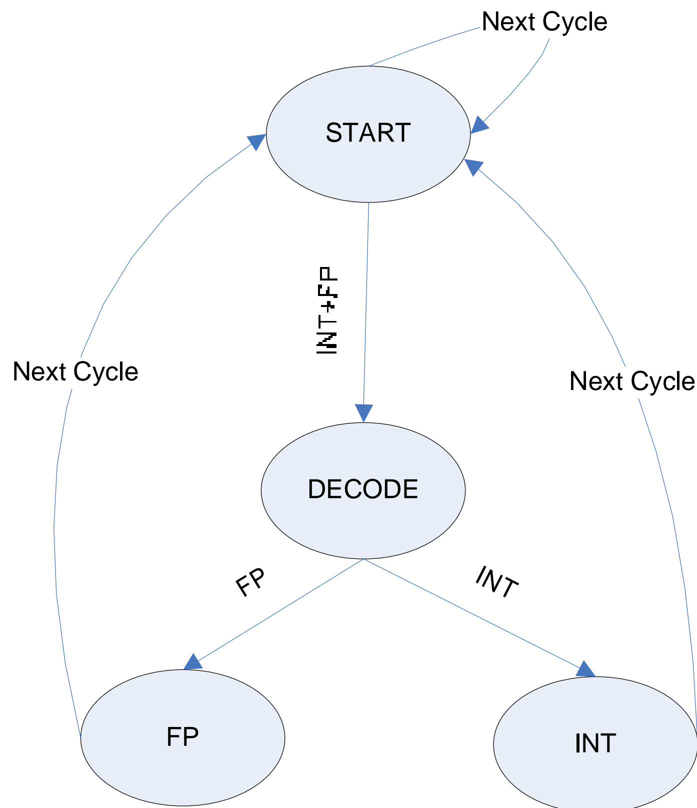
- Integral part of achieving speedup is knowing what the hardware can do
- Only need to be described once per processor
- The compiler can have a generic coding
- Solution must not be memory intensive
- This method is used in GCC
  - But the DFA representation of GCC is very complex.

HMDES Machine Description Language

- Created by John Gyllenhaal in UIUC
- Provides a flexible way to model the pipeline for a variety of processors
- User-friendly yet Powerful
- HMDES is converted to LMDES which is then interfaced with the IMPACT-compiler



GCC Machine Description Example



State Diagram of a Integer & FP Unit Pipeline

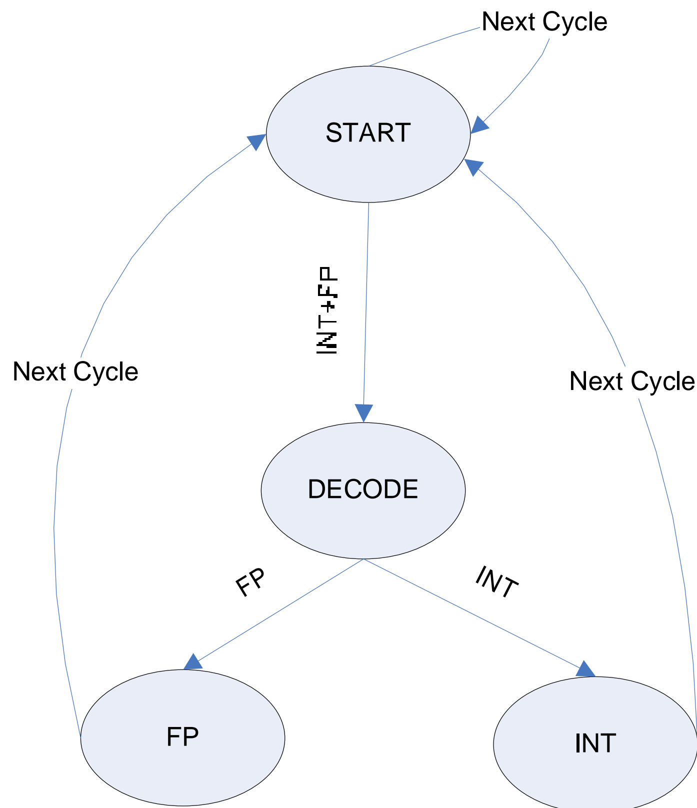
```
(define_cpu_unit "decode")
(define_cpu_unit "int")
(define_cpu_unit "float")

(define_cpu_reservation "int_inst" 2
  (eq_attr "cpu" "int")
  "decode,int")

(define_cpu_reservation "float_insn" 2
  (eq_attr "cpu" "float")
  "decode,float")
```

GCC-DFA representation of this State Diagram

MDES Machine Description Example



State Diagram of a Integer & FP Unit Pipeline

```
CREATE SECTION Resource_Unit
{
  float_insn (use (decode,float));
  int_insn   (use (decode,int));
}

CREATE SECTION Resource_Usage
{
  decode (use (slot1) time(0));
  int    (use (slot1) time(1));
  float  (use (slot1) time(1));
}
```

MDES representation of this State Diagram

Conclusions

- Natural Treegion formation provides significantly larger regions (SEARs extend upon Treegions)
- Application of ICSE to tail duplication – (has the potential for) good tradeoff between code size, compile time, and performance
- What’s missing?

Questions & Contact Information

Chad Rosier mcrosier@ncsu.edu

Tom Conte conte@ncsu.edu

Balaji V. Iyer bviyer@ncsu.edu

TINKER Microarchitecture and Compiler Research Group
North Carolina State University
http://www.tinker.ncsu.edu