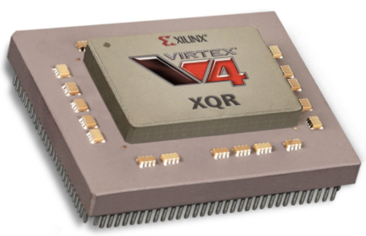
Efficient Software-based Runtime Binary Translation for Coarse-Grained Reconfigurable Architectures

21st Reconfigurable Architectures Workshop
May 19-20, 2014, Phoenix, USA

Toan X. Mai Jongeun Lee
UNIST (Ulsan Nat'l Institute of Science & Tech.), Korea
jlee@unist.ac.kr

Heterogeneous Computing

- Application = kernels + nonkernels
  - kernels: highly parallel, throughput oriented
  - To exploit higher efficiency of accelerators
- Speed-up using accelerator
  - Hardware codecs, FPGA, GPGPU
  - Coarse-grained reconfigurable architecture
- Programming paradigm
  - Traditionally, static & manual partitioning based
  - Need access to application **source code**
  - Need **kernel information**
  - Need knowledge about **accelerator architecture**
  - Manually **partition** and rewrite code for accelerator & CPU



A Xilinx FPGA. source: press.xilinx.com

Compilation for CGRAs

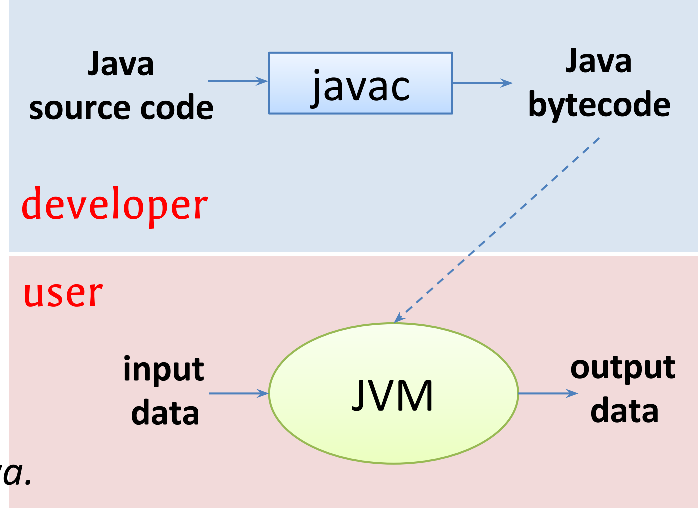
- Programming paradigm is a key challenge
- CGRAs
  - can be easier to program for than FPGAs
    - use of high level language (eg, C)
    - runtime reconfiguration
  - still many problems
    - adoption of new tool flow
    - manual code change
    - no binary compatibility for different CGRA architectures (eg, 4x4 array to 6x6 array)

Our Approach

- Runtime Binary Translation (**RBT**)
  - detect and translate kernels for CGRA at **runtime!**
- Advantages
  - Transparent acceleration
  - Architecture independence
  - **Must-have** in the world of architectural diversity
  - **Free speedup** if you have idle accelerator
  - Can hide runtime translation overhead if SDT is already used

Software Dynamic Translation

- modifies or translates a running program's binary instructions
- eg: Java JIT, valgrind

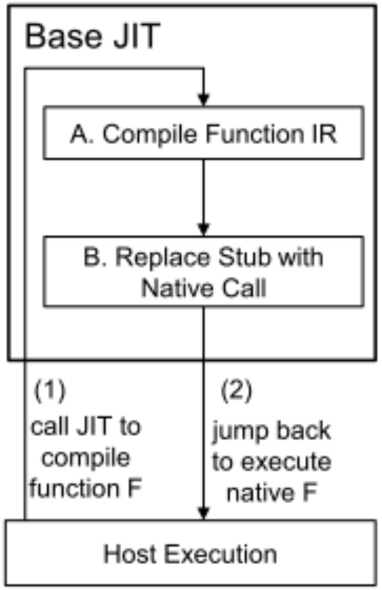


Note: Our current implementation is based on LLVM, not Java.

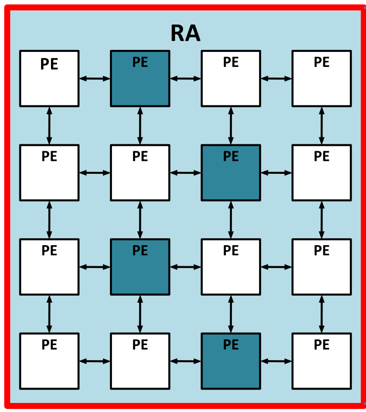
Extending JIT for CGRA

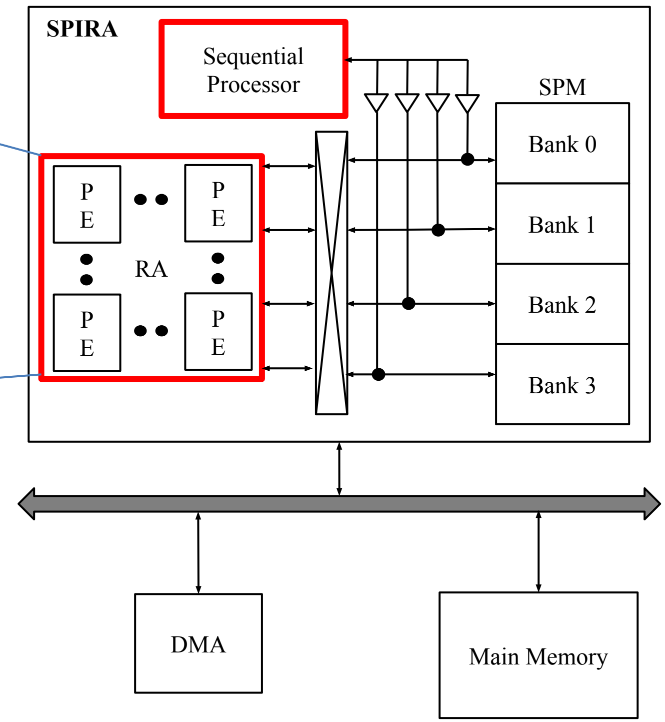
- RBT for CGRA
  - **Discover** suitable kernels by
    - **monitoring** loop execution
    - **examining** loop body
  - **Translate** kernels to accelerator configuration
    - Extract DFG from kernels
    - Map DFG to accelerator using fast compiler
- Challenges
  - kernels are different from functions
    - For functions, translation is a must!
    - For kernels, translation is optional and costly!
  - kernels have input/output data

Just-In-Time compilation: Functions are translated when and only if they are called



Target CGRA Architecture



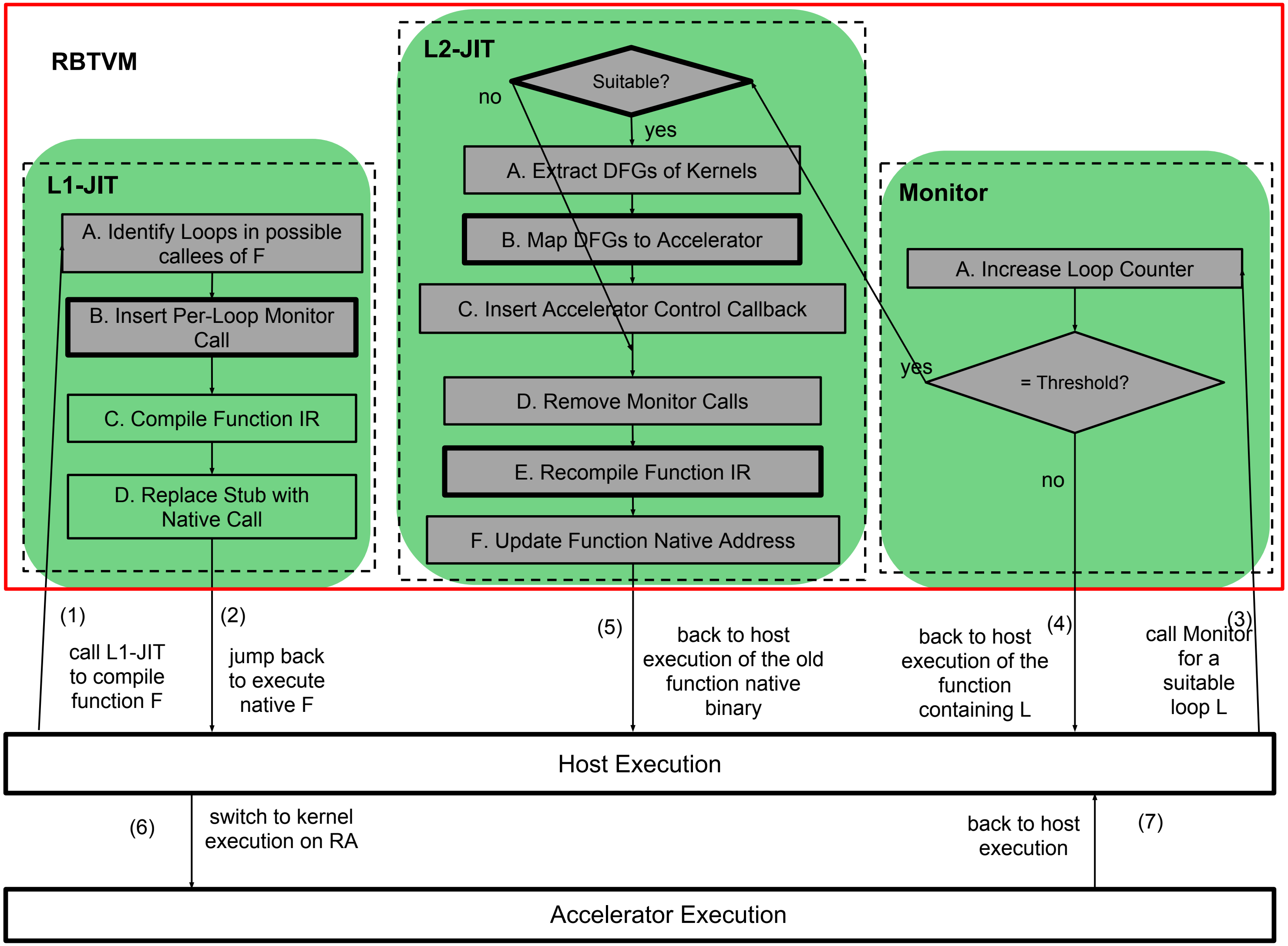


- SP integrated CGRA
  - Sequential Processor (SP)
    - execute seq code, management
  - 4x4 PE array accelerator
    - execute kernels
  - Exclusive execution
  - Shared scratchpad → simplifies data management problem

RBT Challenges

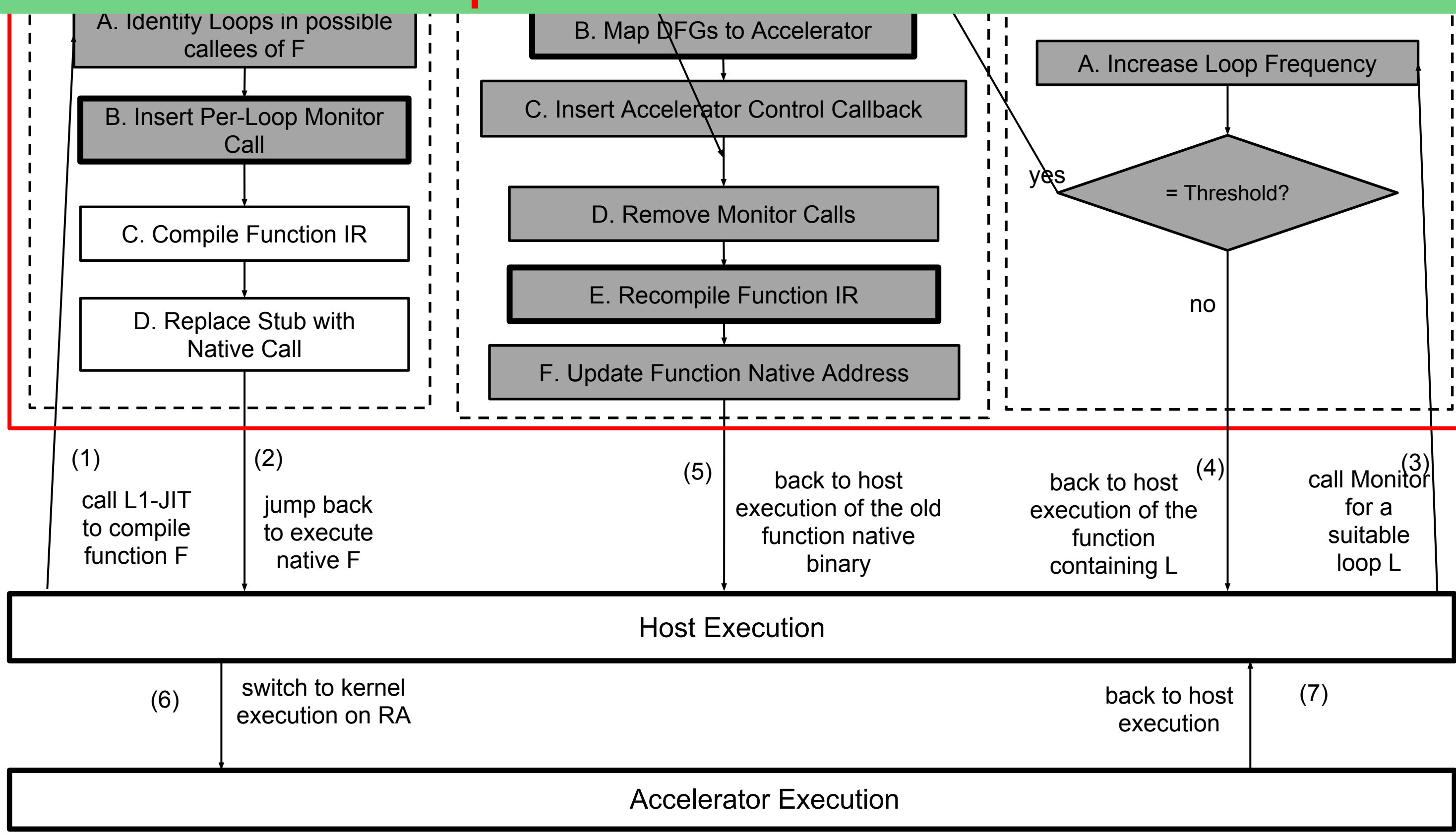
- Objective
  - Maximize **runtime improvement** over the **original JIT**
- First challenge
  - Which kernels to translate
    - Not all loops are kernels worth translating
    - Not all kernels are well-suited for accelerator
      - Contains function calls, or nested loops
      - Variable number of iterations (depending on the arch.)
  - Solution
    - **Monitor** loops to identify kernels
    - **Analyze kernels' body** to determine the suitability

RBT Virtual Machine (RBTVM) Design



Second Challenge

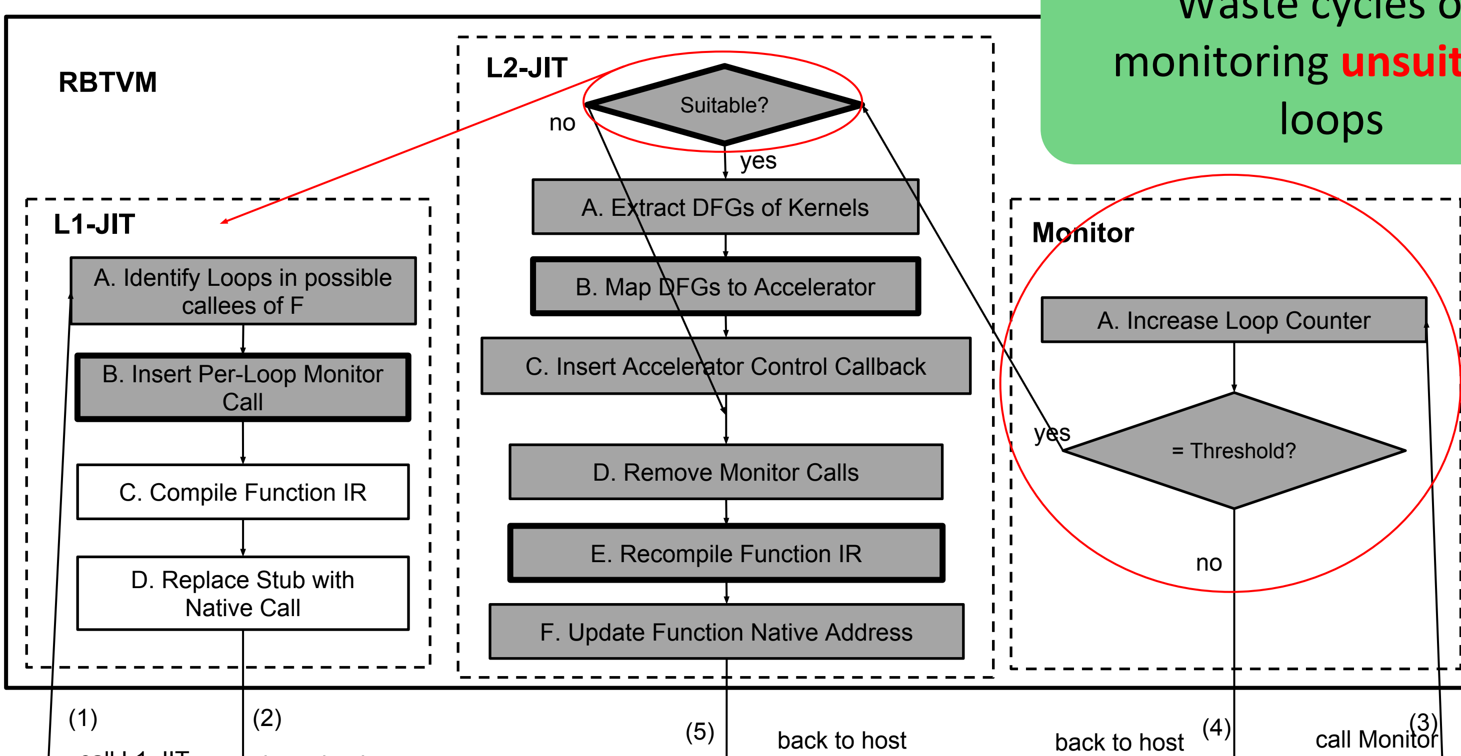
- Second Challenge: Reducing runtime overheads
  - Reduce Monitoring overhead
  - Reduce Recompilation overhead



Reducing Monitoring Overhead

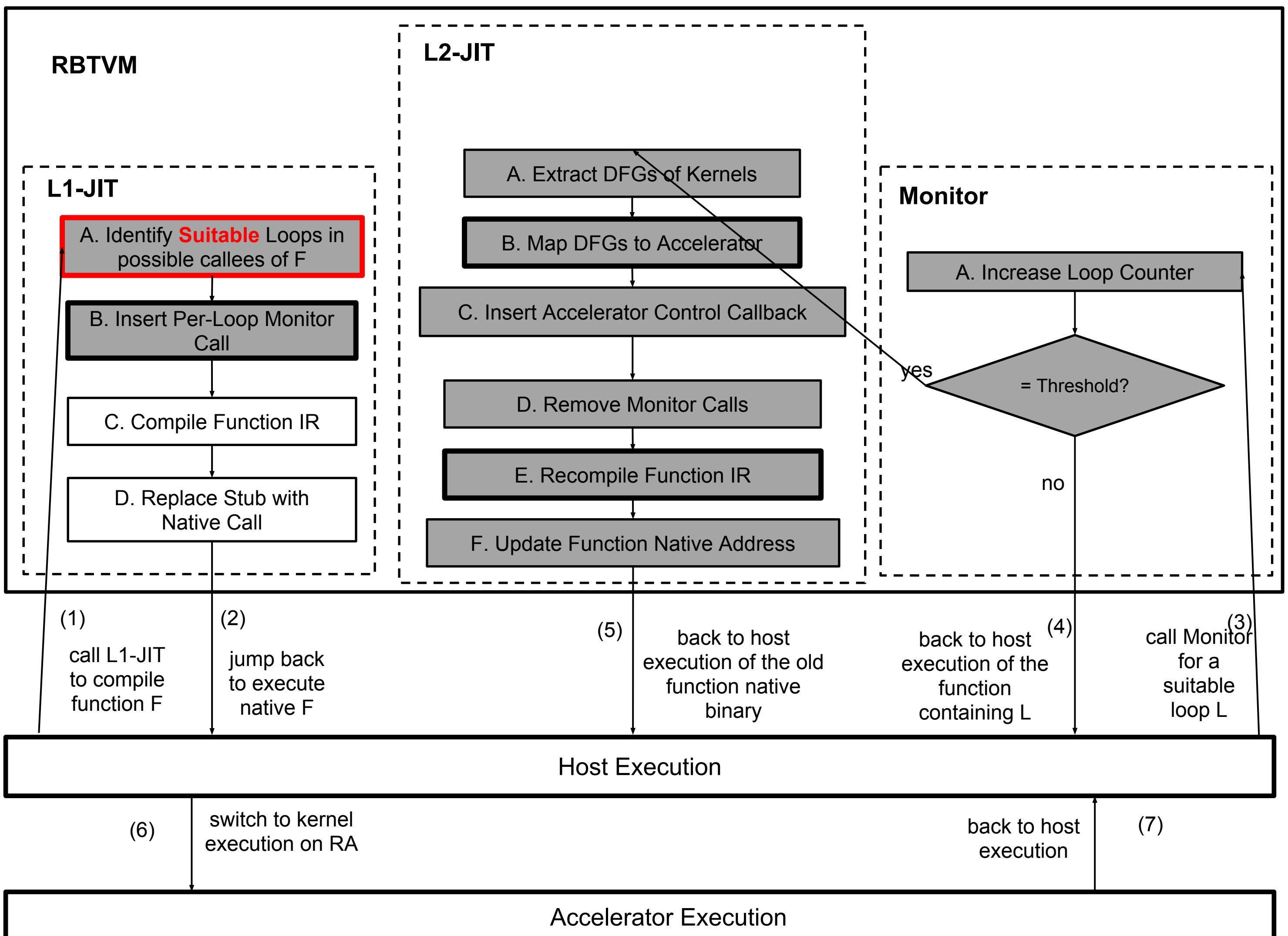
- Unnecessary monitoring for **unsuitable** loops.
- Solution: check suitability in L1 instead of in L2

Waste cycles on monitoring **unsuitable** loops



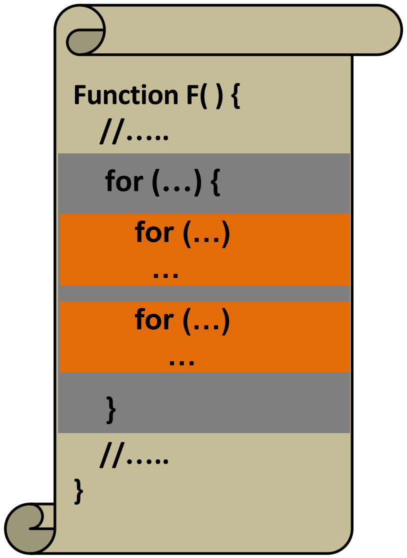
RBT Optimization 1 - Reducing Monitoring Overhead

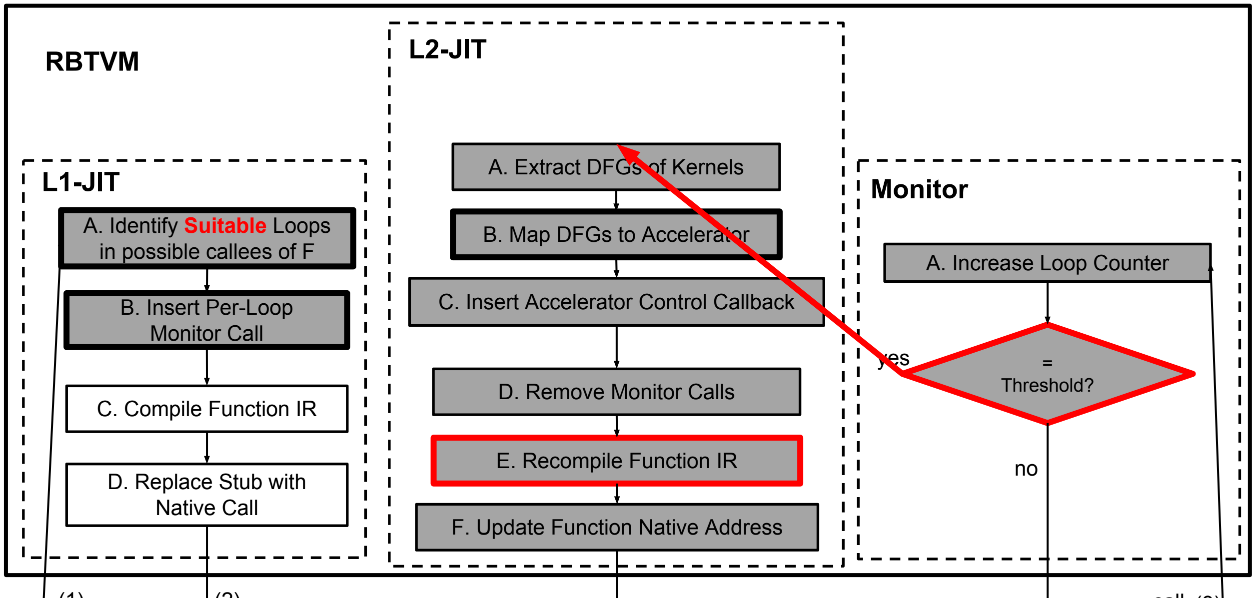
- Check suitability in L1 instead of in L2



Reduce Redundant Recompilation Overhead

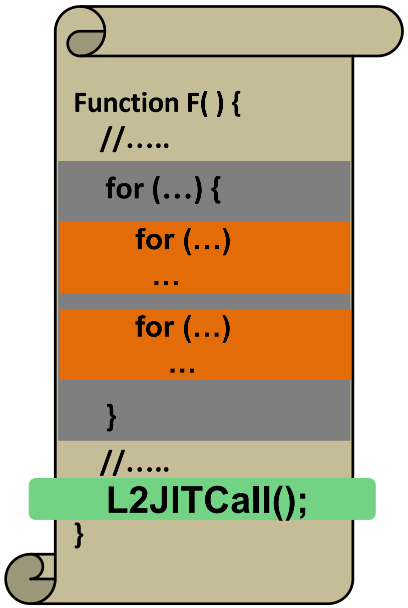
- L2JIT **immediately recompiles** function after kernel is translated
- A function may contain multiple kernels → may be re-compiled several times
  - **2** kernels translated within **same invocation** → **2** recompilations
- Solution: Lazy recompilation
  - Delay the recompilation until the end of the function
  - Do the recompilation just once if multiple kernels are detected within one invocation of the function.

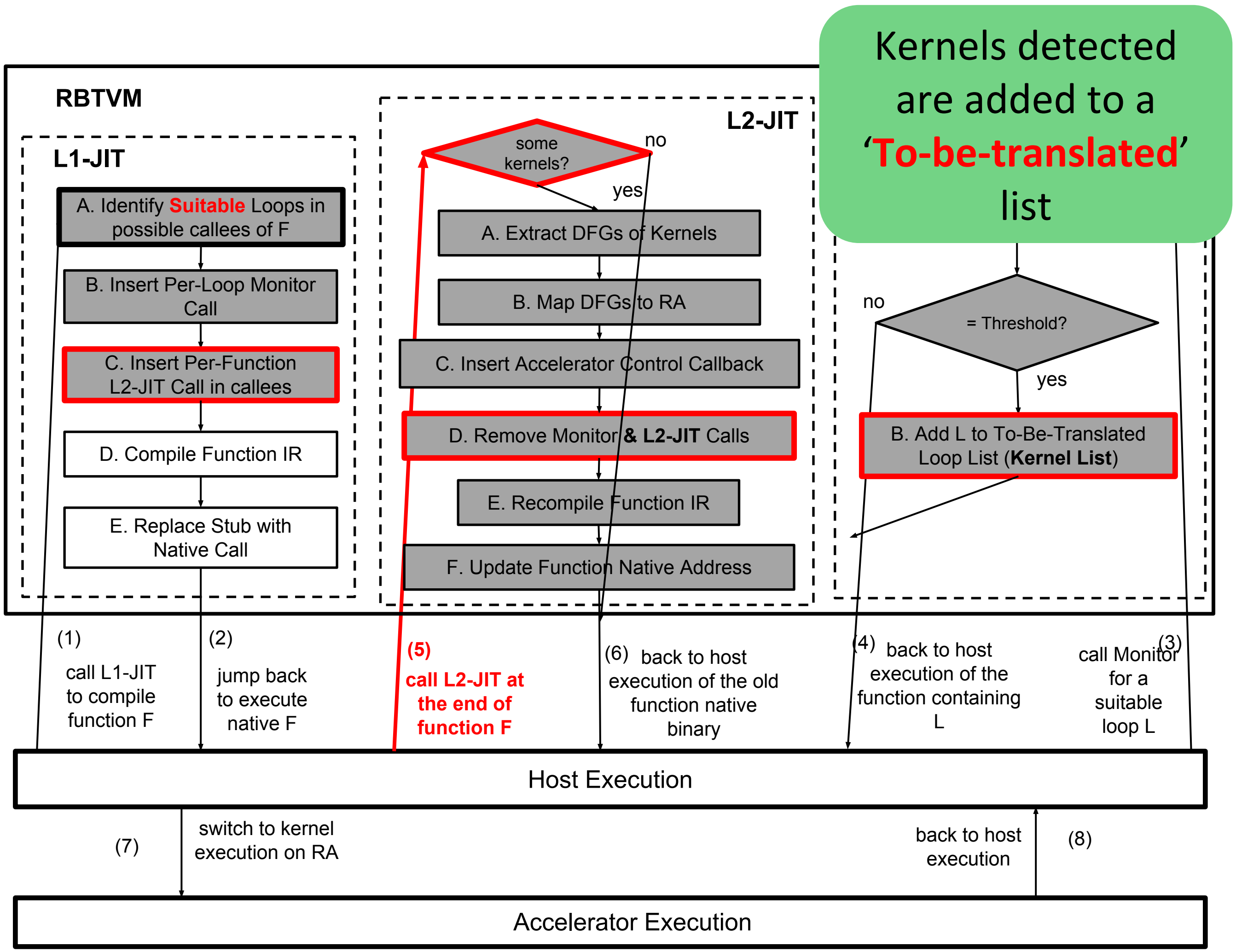




RBT Optimization 2 -
Reducing Redundant Recompilation Overhead

- ❑ **Lazy recompilation**





Implementation

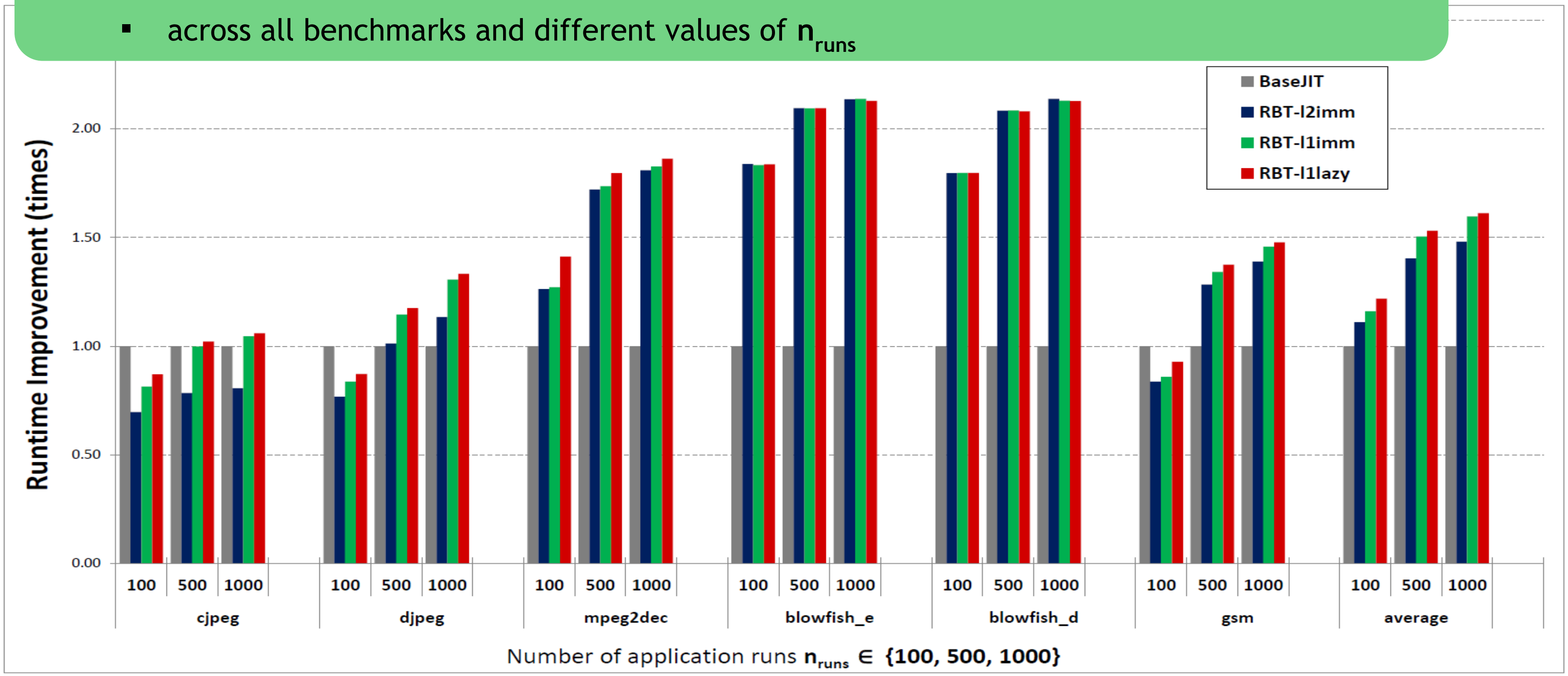
- Implement RBT based on LLVM **JIT**
  - Application first compiled to **LLVM** bitcode (=IR)
  - **lli**: LLVM JIT (=baseline) → extended to support CGRA
  - CGRA mapping algorithm: modulo scheduling
- Performance simulation
  - Modified **lli** running on QEMU PowerPC full system
  - Cycle count: PSIM + DineroIV cache simulator
- Architecture
  - SP: PowerPC running at 400MHz
  - CGRA: 4x4 PE array at 600MHz
  - CGRA cycle count: based on CGRA mapping results
  - Input/output DMA overhead is hidden due to the SP-integrated architecture

Experimental Setup

- Design parameters
  - Kernel Threshold = 50 times execution
    - To qualify as a **kernel**
  - Threshold for **# Operations of Kernel** = 80 ops
    - To determine if a kernel should be translated
    - Otherwise too much time on kernel translation
- Benchmarks
  - **MiBench** (cjpeg, djpeg, blowfish e/d, gsm)
  - **MediaBench** (mpeg2dec)
  - Compiled using Clang (**-loop-simplify**, **-indvars**, etc.)
- Evaluation criteria
  - Runtime improvement
  - Overheads evaluation (effectiveness of our optimizations)
  - Vary # of app. runs (n_{runs}) for each app to **{100, 500, 1000}**

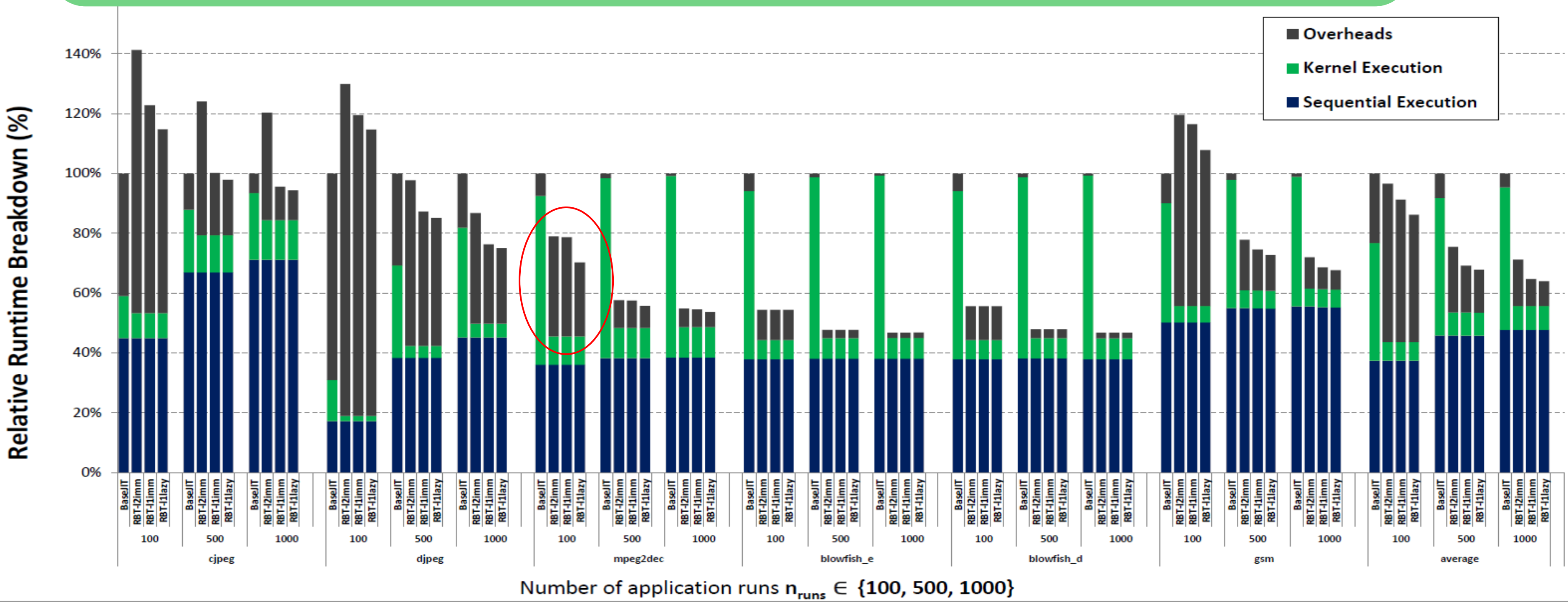
Runtime Improvement

- 4 cases
  - BaseJIT (baseline): original JIT, without accelerator support
  - RBT-l2imm: **RBTVM** without any optimization
  - RBT-l1imm: RBTVM + **1st** optimization (checking suitability in L1 instead of in **L2**)
  - RBT-l1lazy: RBTVM + **1st + 2nd** optimization (lazy recompilation instead of **immediate**)
- Better runtime improvement as n_{runs} increases
- RBT-l1lazy gives best average improvement: 1.44 times over **BaseJIT**
  - across all benchmarks and different values of n_{runs}



Runtime Breakdown

- 4 cases: BaseJIT, RBT-l2imm, RBT-l1imm, and RBT-l1lazy
- Runtime = **Seq.** Exec + **Kernel** Exec + **Overheads**
- **BaseJIT runtime** represents **100%**
- **Kernel Speedup** is the **main factor** leading to Runtime improvement
- Average Kernel Speedup (for all three RBT cases): **5.88 times**
- Three RBT cases **differ** in the **overheads**

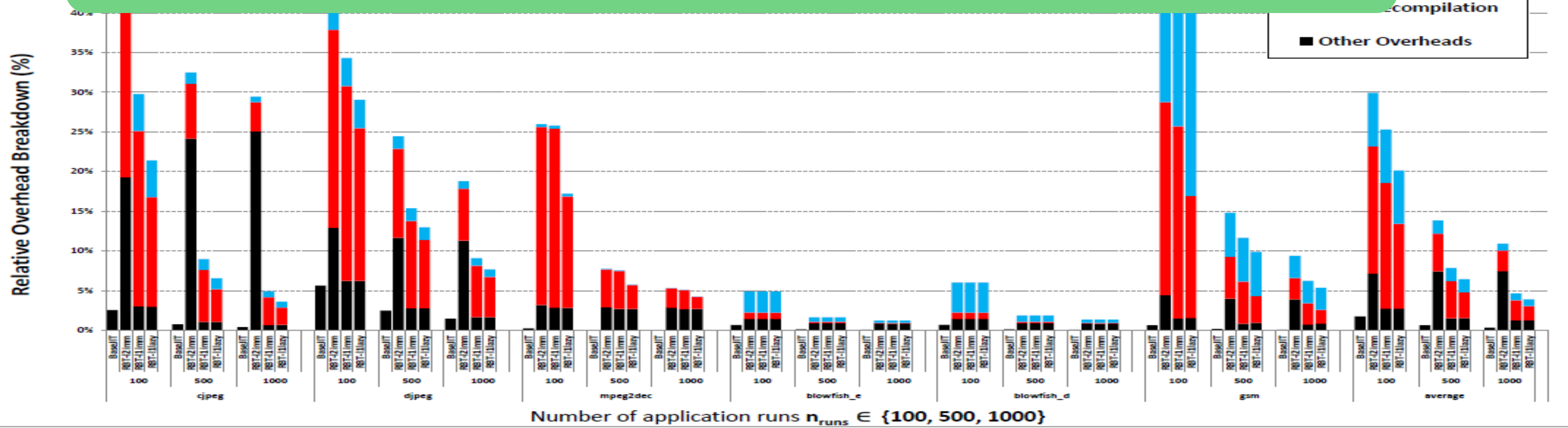


Overhead Breakdown

- Overheads = Kernel Translation + **Recompilation** + **Other Overheads**
- **Other Overheads = Monitoring + Context Switching**
- 4 cases: BaseJIT, RBT-l2imm, RBT-l1imm, and RBT-l1lazy
- **BaseJIT runtime** represents **100%**

- **+ 1st optimization: RBT-l2imm → RBT-l1imm:**
  - 75.00% reduction in **Monitoring** & **Context Switching** overheads are reduced

- **+ 2nd optimization: RBT-l1imm → RBT-l1lazy:**
  - 31.94% reduction in Recompilation **Overhead**



Related Work

- Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications (Beck et al., DATE'08)
- Application of Binary Translation to Java Reconfigurable Architectures (Beck et al., IPDPS'05)
  - Require additional hardware for binary translation and analyzing instruction sequence
  - Very different accelerator architecture
- A Java Virtual Machine for Runtime Reconfigurable Computing (Greskamp et al, FCCM'05)
  - Targeting FPGA accelerators
- **Differences compared with the previous work**
  - Pure software solution (JIT-based VM only)
  - Target architecture: equipped with CGRA accelerator (2D array of PEs)

Conclusion

- **New programming paradigm**: runtime translation
  - Can greatly enhance usability of reconfigurable processors
  - Also enables exploiting smaller kernels
  - Shared scratchpad memory is essential
- In this paper
  - Propose **RBT VM** design
  - **Optimizations** and **implementation** for RBT VM
  - Over **50% speedup at application level** can be achieved
- Future Work
  - Investigate tradeoff between translation time vs. mapping quality
  - Compare with other runtime compilation approach (eg, OpenCL)
  - Extension for more general architecture (without shared memory)