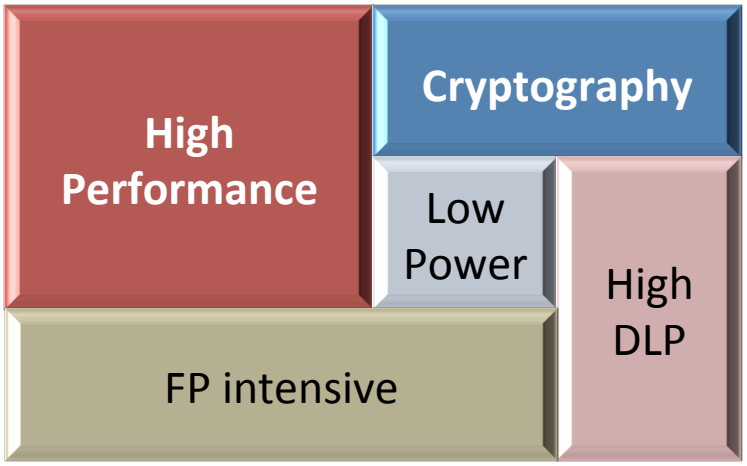
Execution Migration in a Heterogeneous-ISA Chip Multiprocessor

Matthew DeVuyst Ashish Venkat Dean M. Tullsen

University of California, San Diego

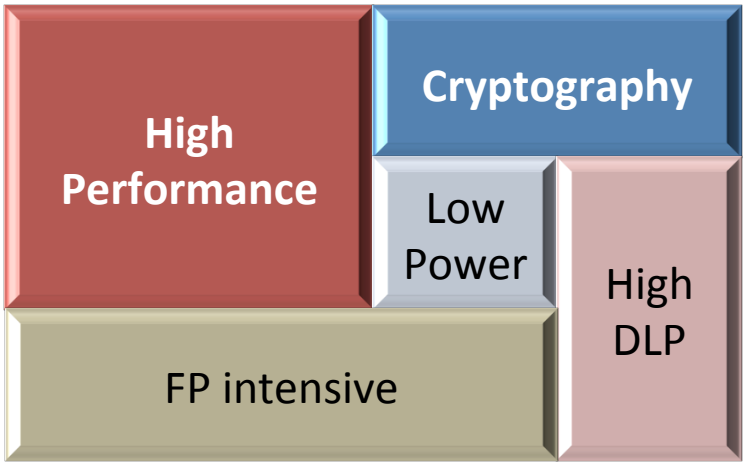
Heterogeneous multi-core processors

- Could be composed of both high-performance power-hungry cores and low-performance power-efficient cores
- Each core could be specialized for a different class of application
- Application could migrate b/w cores during different phases of execution



Heterogeneous multi-core processors

- Prior research has shown that a single-ISA heterogeneous multicore processor can
  - Outperform a homogeneous one by about 63%*
  - Achieve upto 69% energy savings with only 3% drop in performance*
- However, that research restricts cores to a single ISA to avoid issues during migration
- Is that really a necessary constraint? Because reducing the cost of migration could eliminate this restriction.



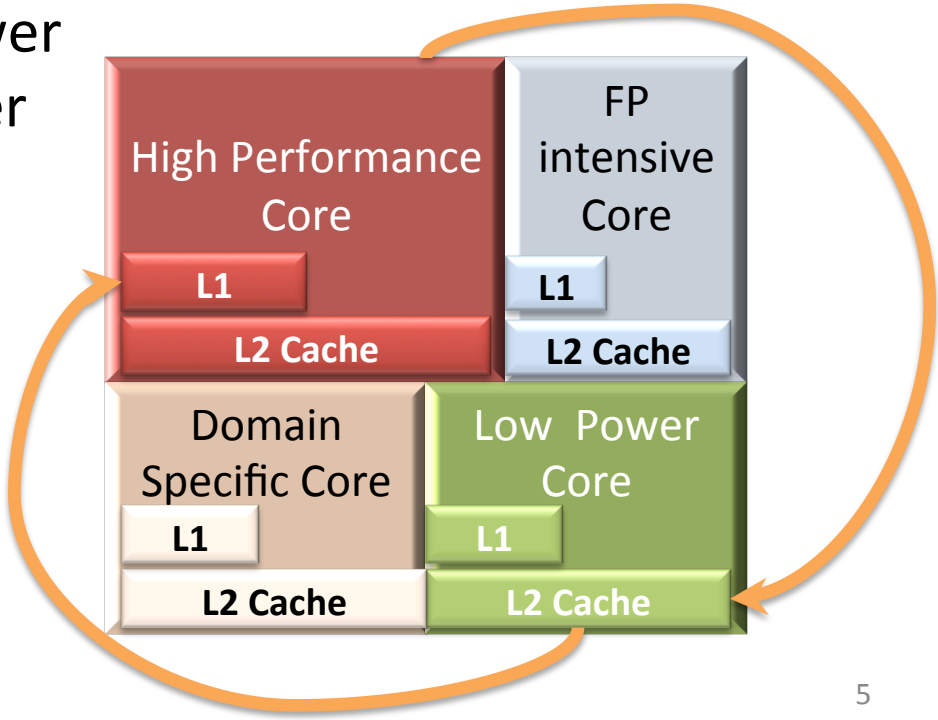
* Rakesh Kumar, Keith Farkas, Norm P. Jouppi, Partha Ranganathan, Dean M. Tullsen, MICRO'03

Our contention is. . .

- **Restricting cores to a single ISA eliminates an important dimension of heterogeneity**
- ISAs are designed for different goals:
  - High performance (x86)
  - Energy efficiency (ARM)
  - Reduced code size - Thumb ISA consumes 30% less power due to code compression
  - Domain specific instructions
  - Compute bound vs memory bound?
  - ILP vs DLP?

However . . .

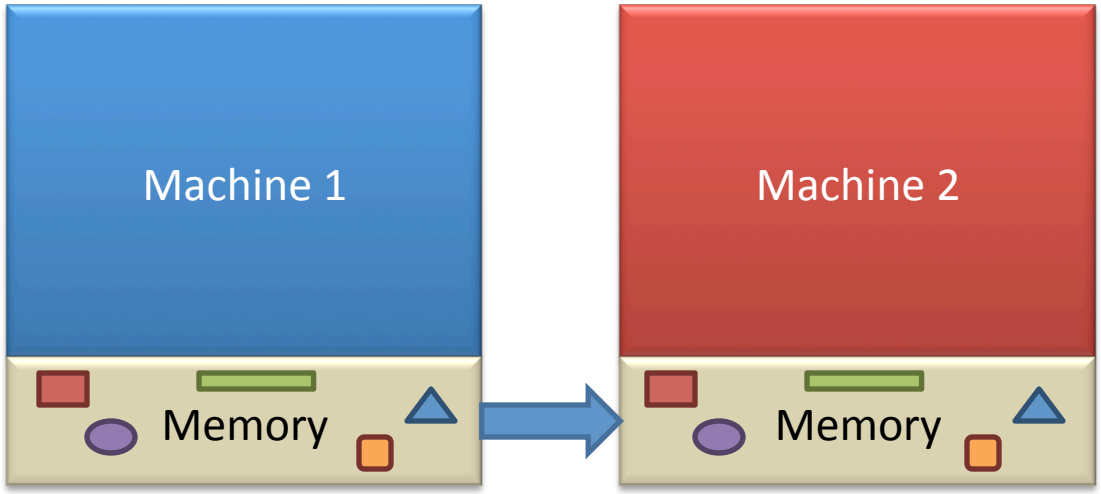
- **Execution migration is important in a heterogeneous multicore processor**
  - To find the best possible core for an application or a section of an application
  - To move processes to a lower power core when the power cord is plugged out
  - To move applications to cooler parts of the system once a thermal threshold is reached
  - To perform load balancing



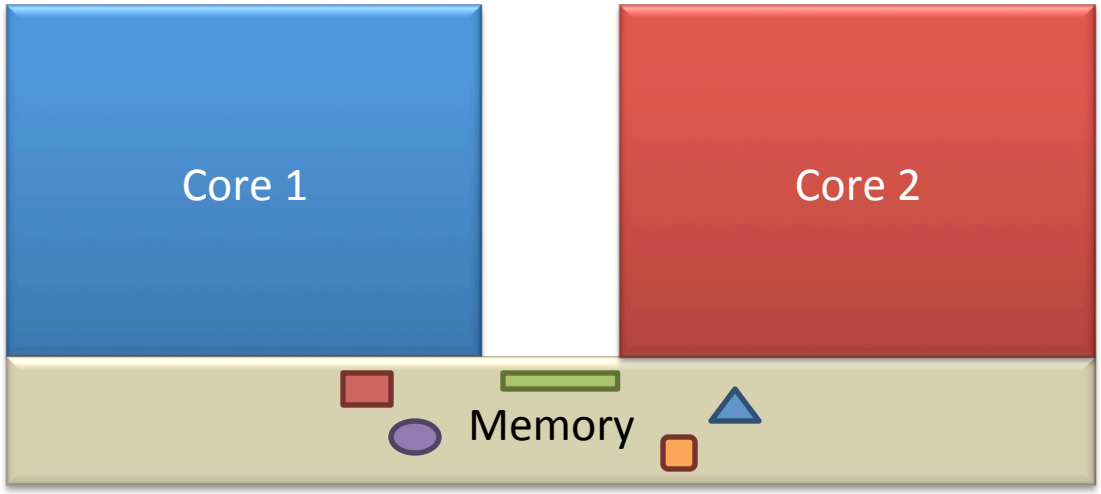
Why is migration a hard problem?

- Transfer of memory image
- Transformation of architecture-specific program state
  - Reorder objects in memory
  - Fix pointers
- Creating register state

Opportunity for on chip heterogeneity



Memory image transfer
State transformation
Total time:
140 + 20 = 160 ms*



State transformation
Total time: 20 ms

Two implications here

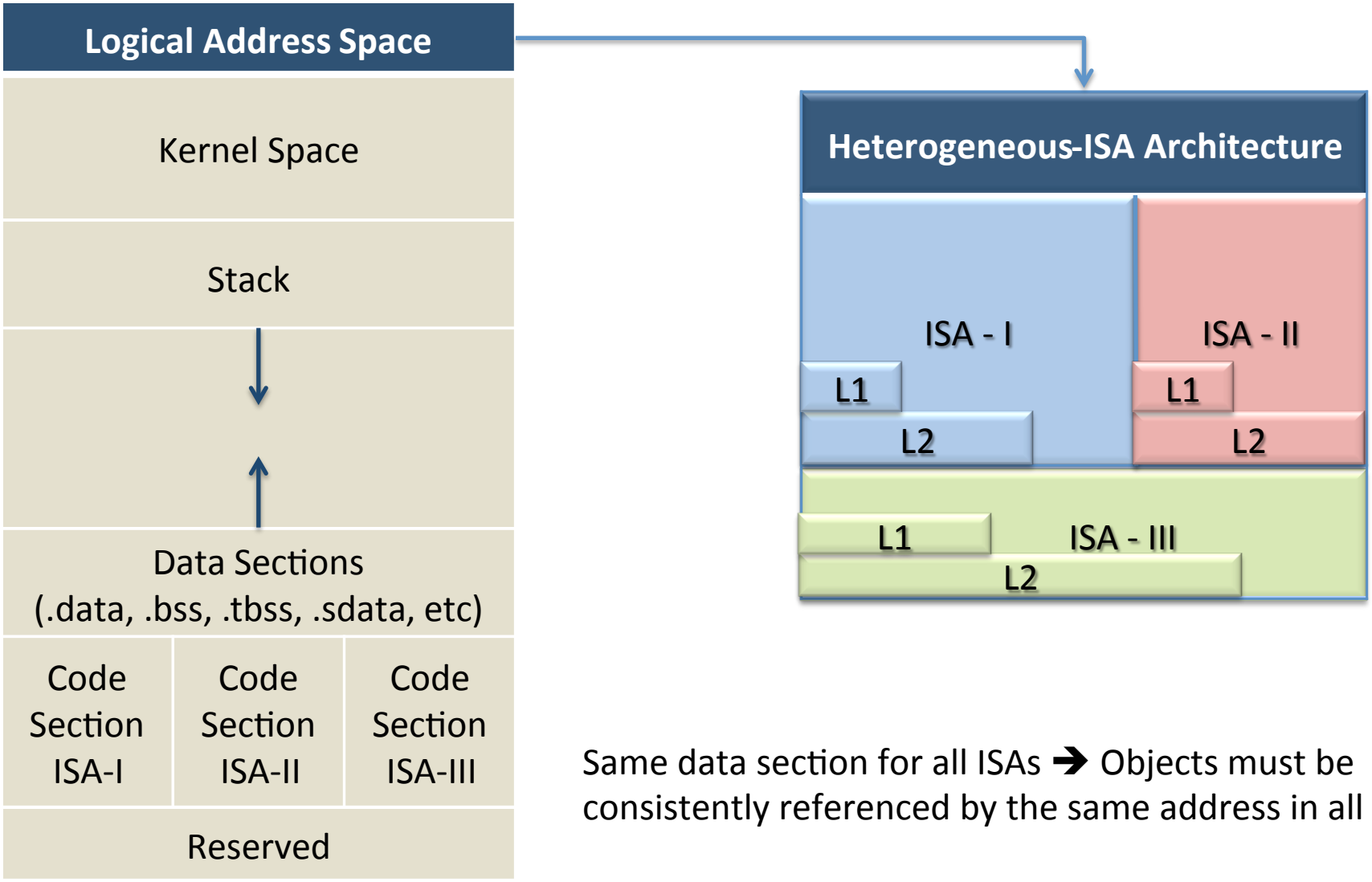
- **Very fast migration**
- **State transformation cost is highly exposed**

* The Tui System, University of British Columbia, March 1997

Outline

- Motivation
- Our strategy
- Compiling for heterogeneous-ISA architecture
  - Memory image consistency
- Program State Transformation
  - Overview of migration
  - Stack transformation
  - Binary translation
- Conclusion

Our strategy



Same data section for all ISAs ➔ Objects must be consistently referenced by the same address in all ISAs

Memory image consistency

- Data section consistency
  - Ensure objects are placed at the same location for all ISAs
- Code section consistency
  - Ensure function pointers point to the same function for all ISAs
- Stack consistency
  - Ensure objects on the stack are consistently placed for all ISAs

Optimize steady state performance or Program transformation cost?

Static Compilation Low performance impact	Program Transformation Low transformation cost
Place global variables and Functions at the same address - Program transformer doesn't need to reorder global objects	Transform register state from one ISA to another - Allows compiler to use efficient register allocation strategies
Place objects whose addresses are taken at known locations - Program transformer doesn't need to fix pointers	Transform stack frames to use conventions of other ISA - Allows compiler to use efficient register allocation strategies

At what points can we migrate?

- At every instruction?
  - Object placement is uniform across all ISAs in all sections
  - Requires no transformation at runtime
  - Heavily sacrifices architecture-specific compiler optimizations
- At specific points of equivalence
  - Objects are consistently placed with minimal transformation
  - We use function call sites as points of equivalence
  - To support instantaneous migration, binary translation is performed till a point of equivalence is reached

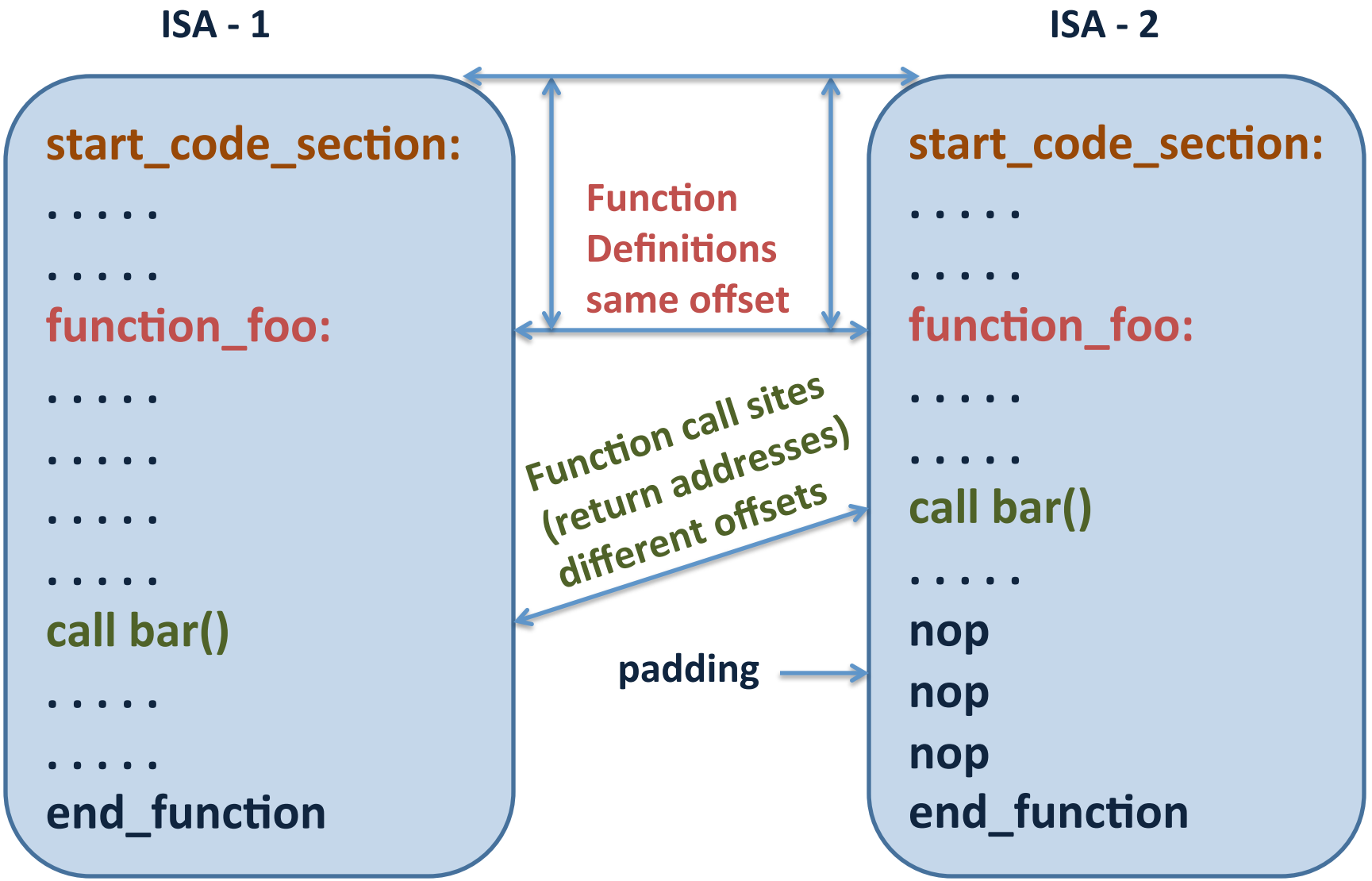
Outline

- Motivation
- Our strategy
- Compiling for heterogeneous-ISA architecture
  (Memory image consistency)
  - Data section consistency
  - Code section consistency
  - Stack consistency
- Program State Transformation
  - Overview of migration
  - Stack transformation
  - Binary translation
- Conclusion

Data section consistency

- Ensure consistent endianness and basic data type
- For each global data section, ensure that the following are consistent across ISAs
  - Number of objects
  - Size of each object
  - Relative order in memory
  - Alignment and padding rules
- Ensure dynamic memory allocation gives the same virtual address regardless of the ISA

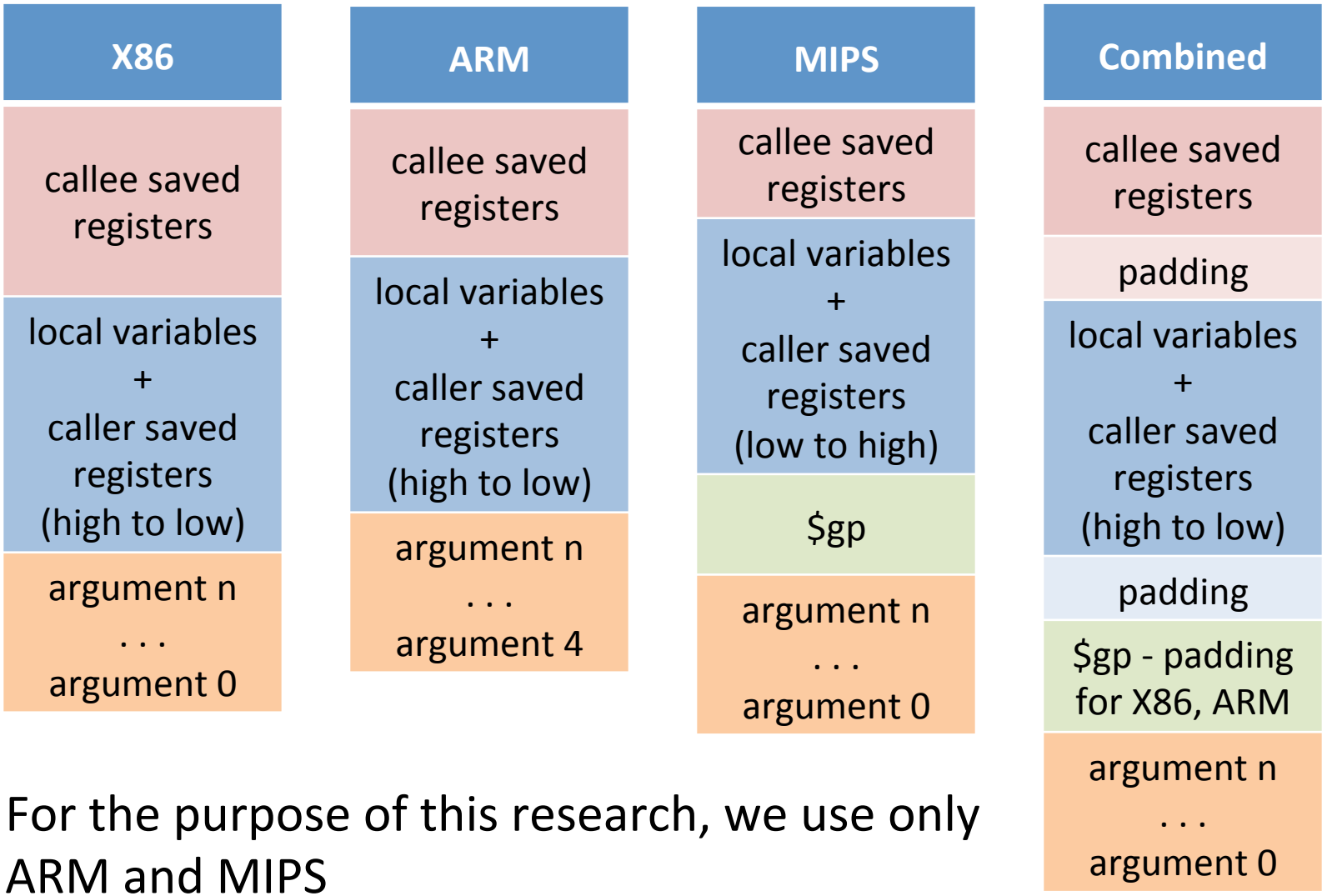
Code section consistency



Stack Consistency

- Stack interaction is carefully optimized for each ISA. Most objects on the stack would have to be moved at runtime.
- To minimize the number of transformations, ensure that the following are consistent
  - Direction of stack growth
  - Size of each stack frame
  - Offset of different regions of a stack frame from the frame pointer (Add padding where necessary)
  - Alignment of objects in a stack frame
- Allocate large aggregate objects and objects whose addresses are taken at the beginning of a region

Stack Consistency – an example

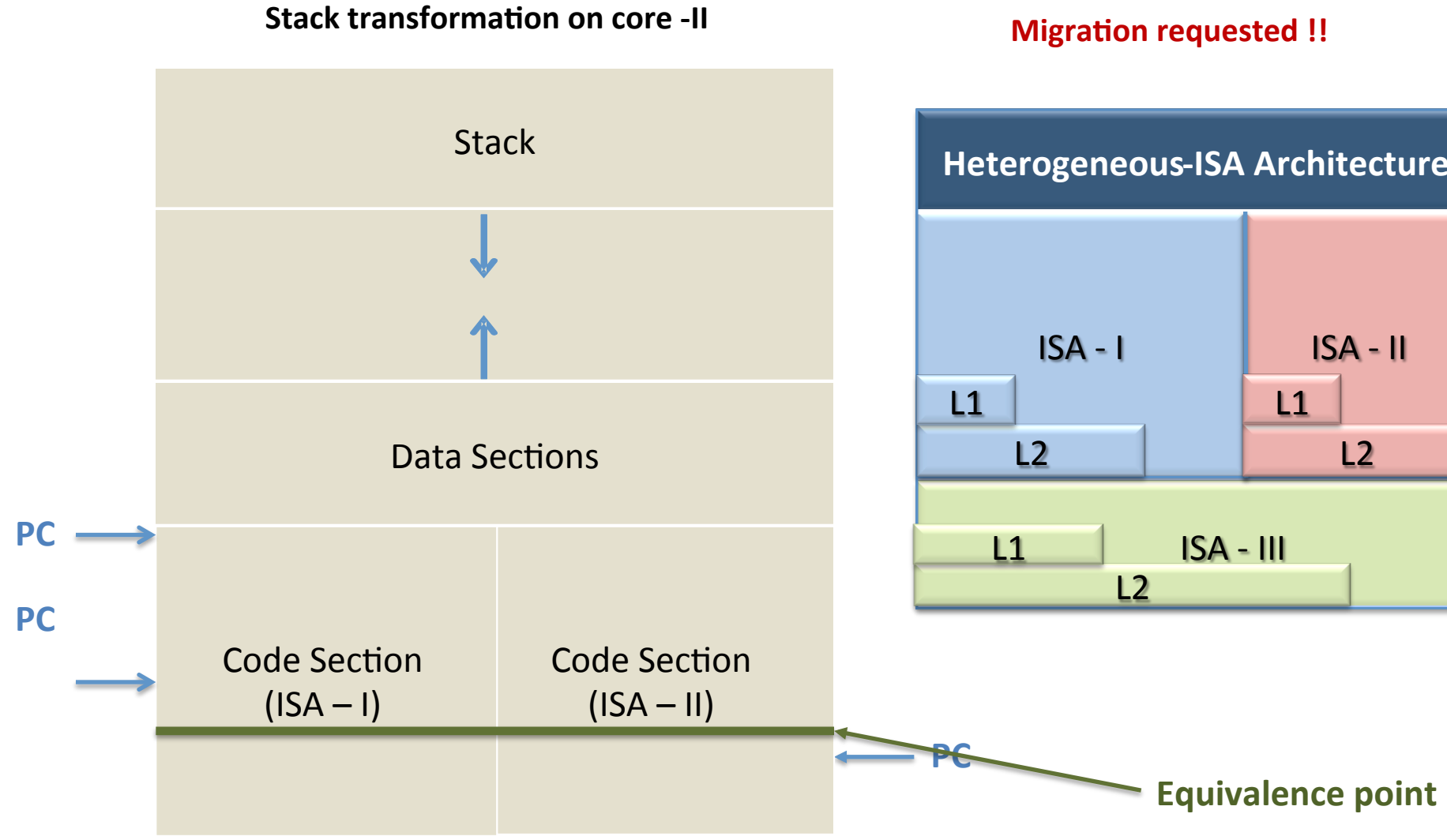


For the purpose of this research, we use only ARM and MIPS

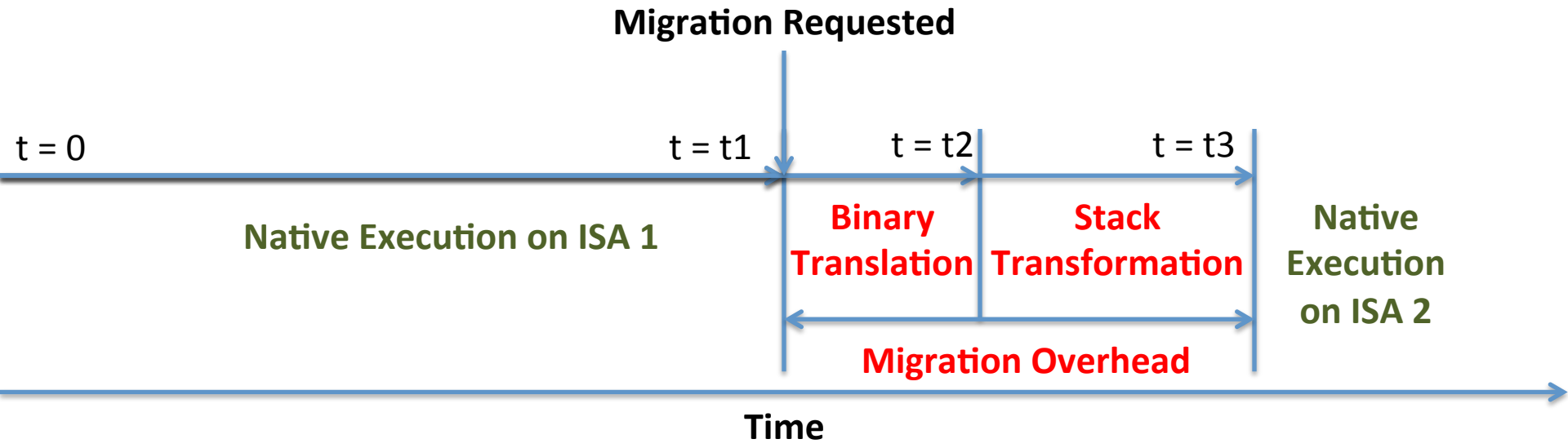
Outline

- Motivation
- Our strategy
- Compiling for heterogeneous-ISA architecture
  - Memory image consistency
- Program State Transformation
  - Overview of migration
  - Stack transformation
  - Binary translation
- Conclusion

Overview of migration



Execution Migration Timeline



Migration Overhead = Binary Translation + Stack Transformation

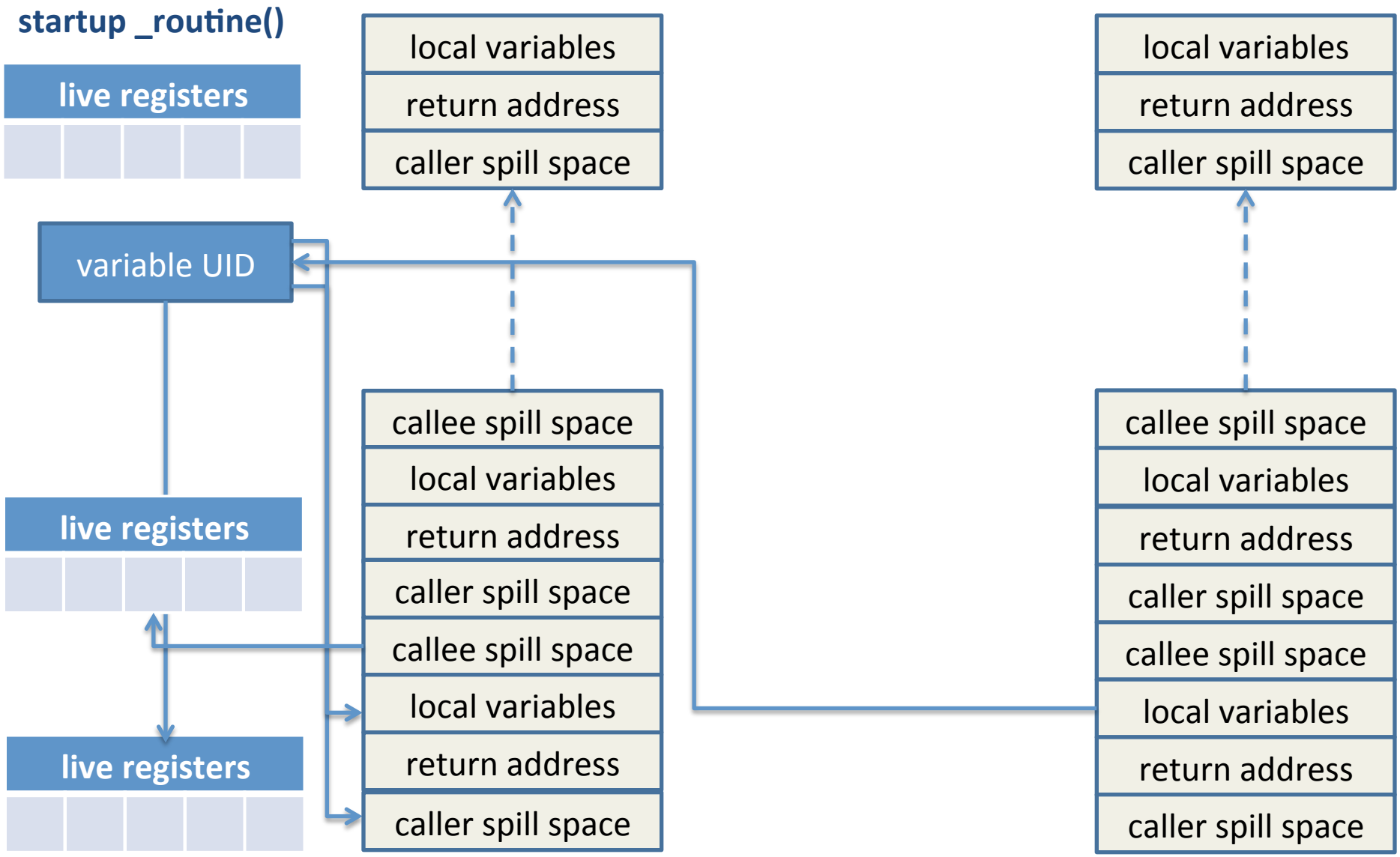
Outline

- Motivation
- Our strategy
- Compiling for heterogeneous-ISA architecture
  - Memory image consistency
- Overview of migration
- Stack Transformation
  - Mechanisms
  - Results
- Binary Translation
  - Mechanisms
  - Results

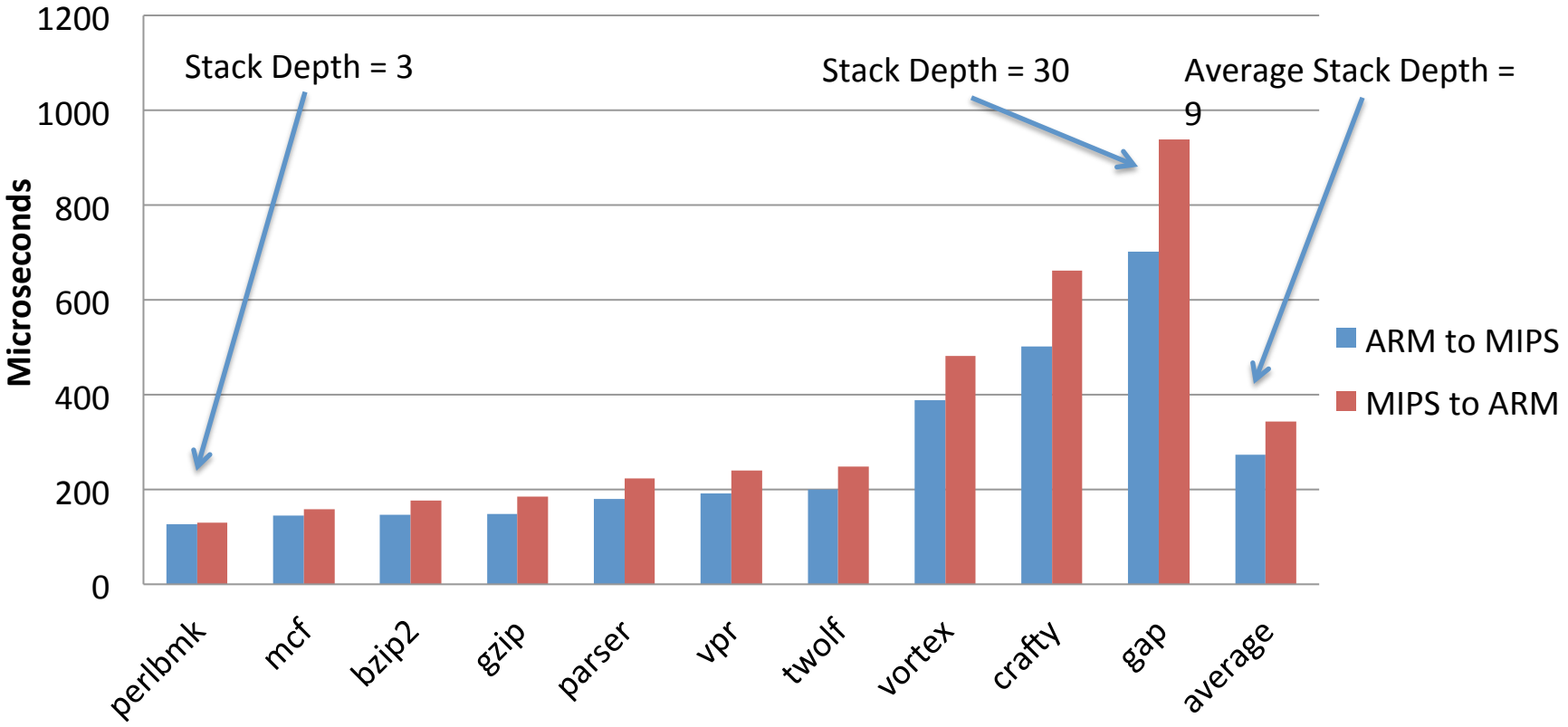
Stack Transformation

- Goal of stack transformation
  - To move values of local variables in open function activations to the right stack offsets
  - To fix all return addresses
  - To create register state for migrated-to core
- To perform this, we collect the following information during compilation for each ISA
  - Frame layout for each function
  - Function call site details
  - Location of variables at each function call site
  - Sets of spilled caller and callee saved registers
  - List of live registers across each function call site

Stack Transformation – An example

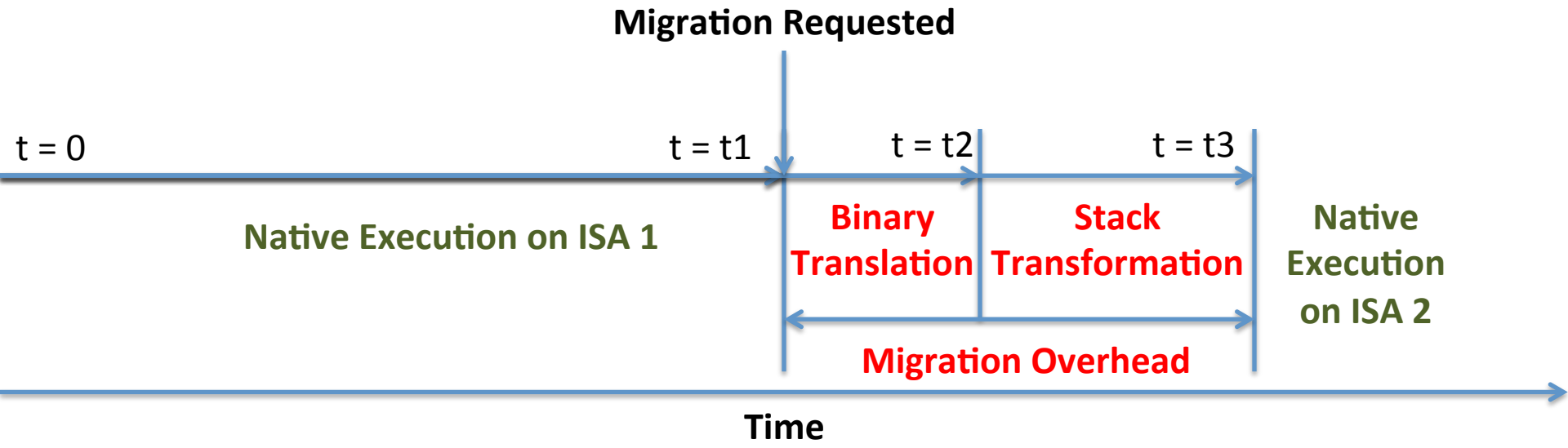


Stack Transformation Costs



- Directly proportional to number of frames processed (stack depth)
- Average Stack Transformation cost = 300 µs
- Copy + Transformation costs from prior research = 160 ms
- > 500X Speedup

Execution Migration Timeline



Migration Overhead = Binary Translation + Stack Transformation

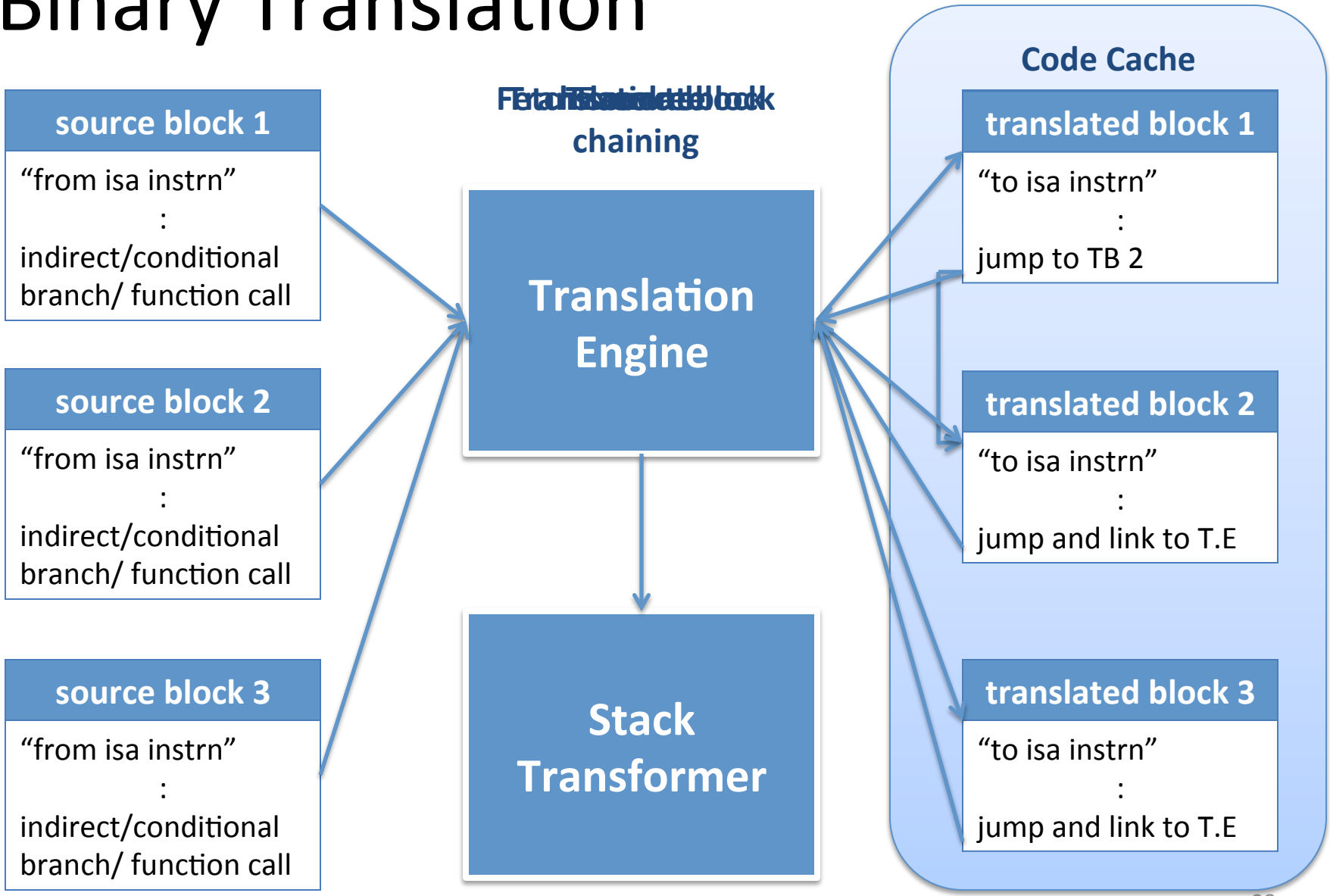
Outline

- Motivation
- Our strategy
- Compiling for heterogeneous-ISA architecture
  - Memory image consistency
- Overview of migration
- Stack Transformation
- Binary Translation
  - Mechanisms
  - Results
- Conclusion

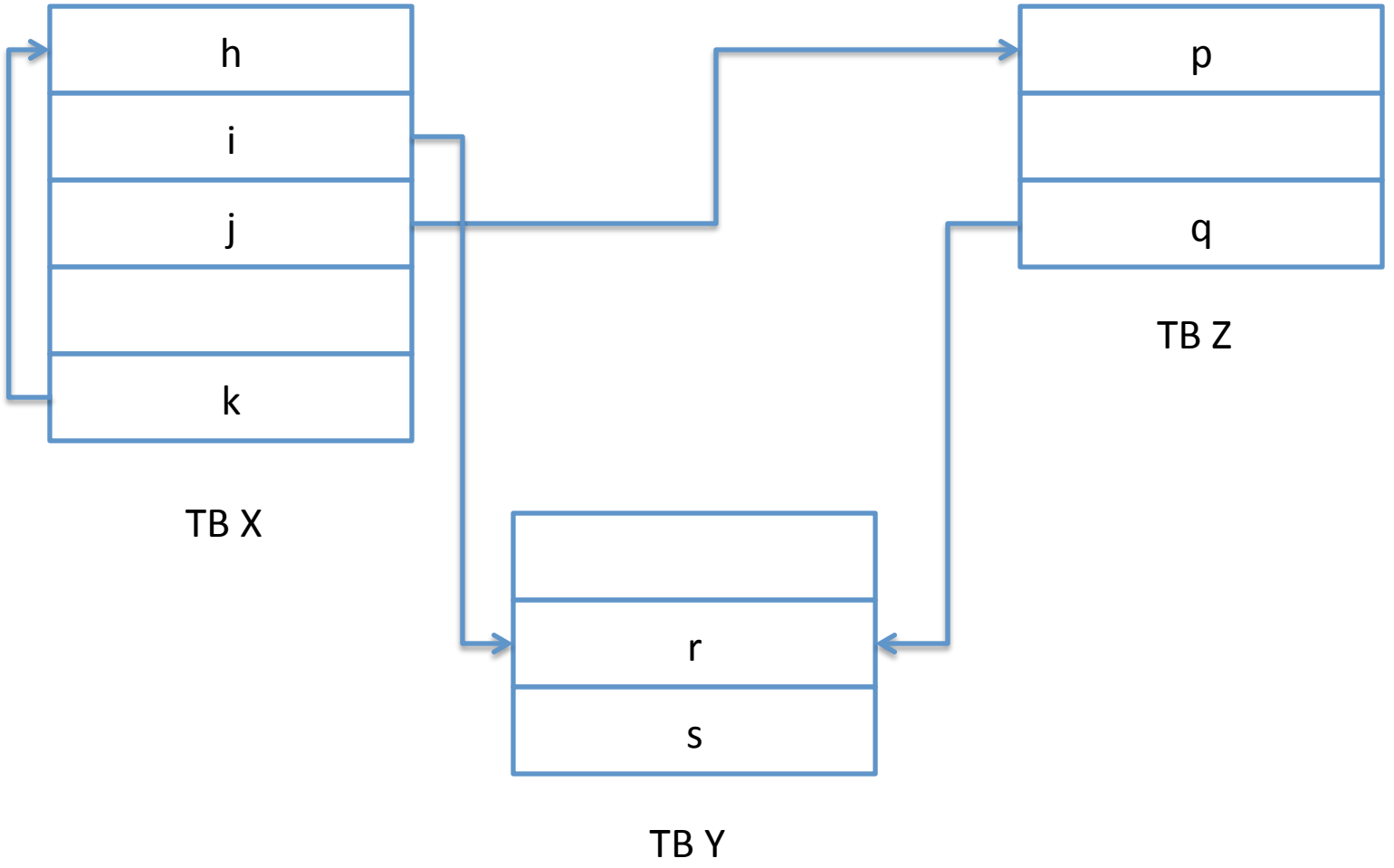
Binary Translation

- Facilitates instantaneous migration
- Classic JIT dynamic translation is performed at the point of migration till an equivalence point (function call) is reached
- Conventional binary translators execute trillions of instructions
- We execute a smaller number of instructions at the time of migration
- **Our binary translator is optimized for the migration use case**

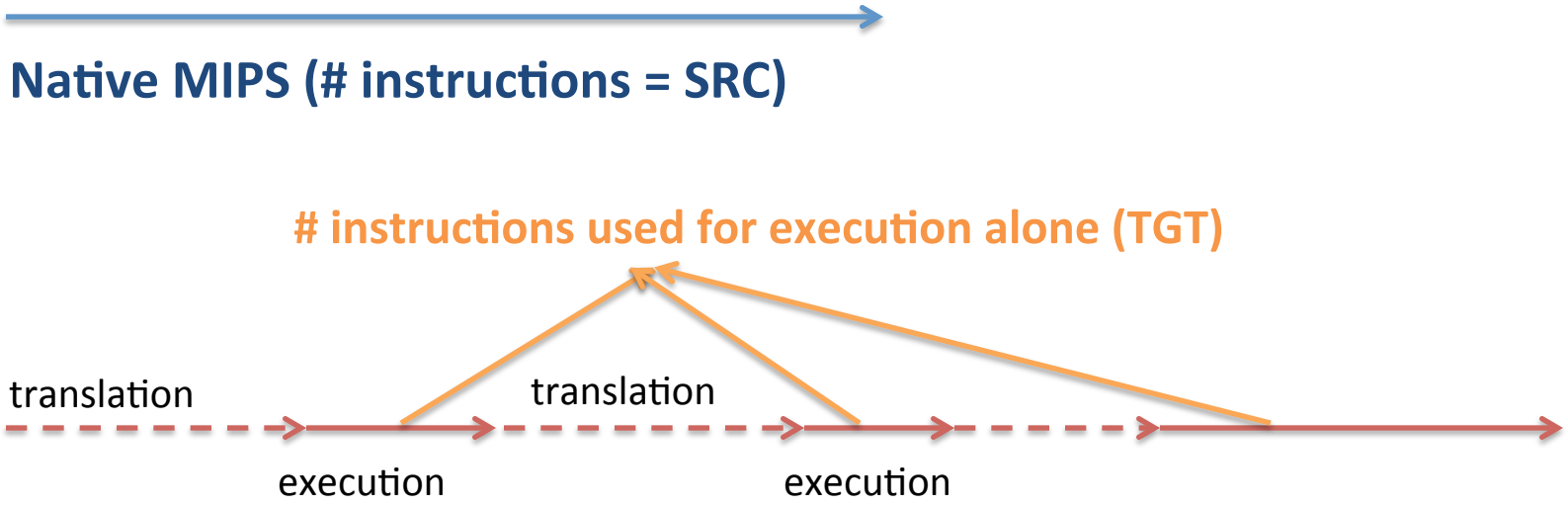
Binary Translation



Multiple-entry Multiple-exit Translation Block Chaining



Performance Metrics

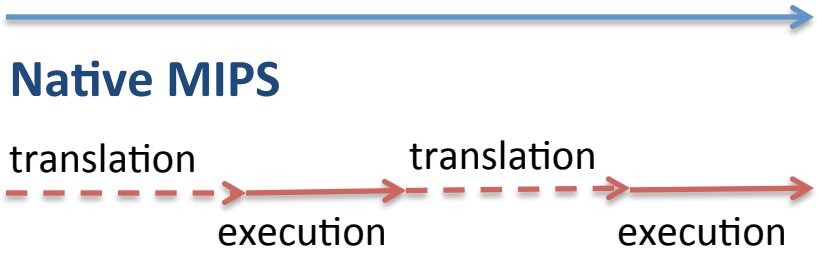


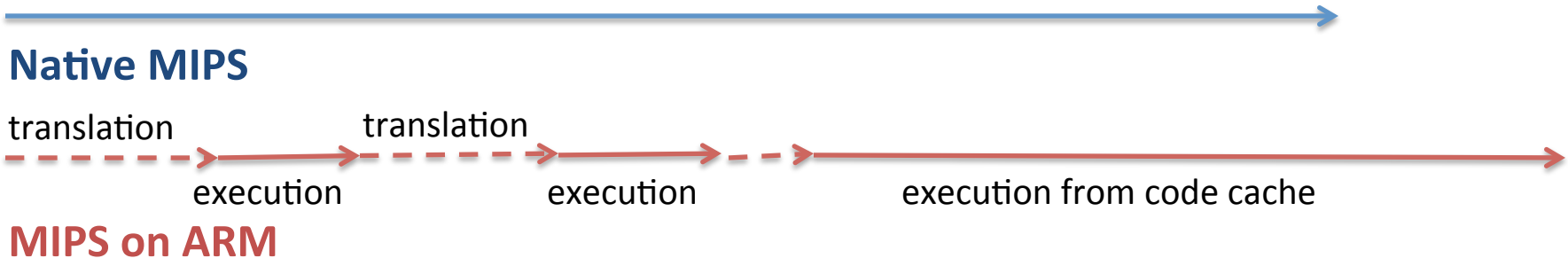
We measure two ratios

- Total to Source Ratio (**TOTAL/SRC**) – Includes translation costs
- Target to Source Ratio (**TGT/SRC**) – Excludes translation costs

Both these ratios must be as low as possible for best performance

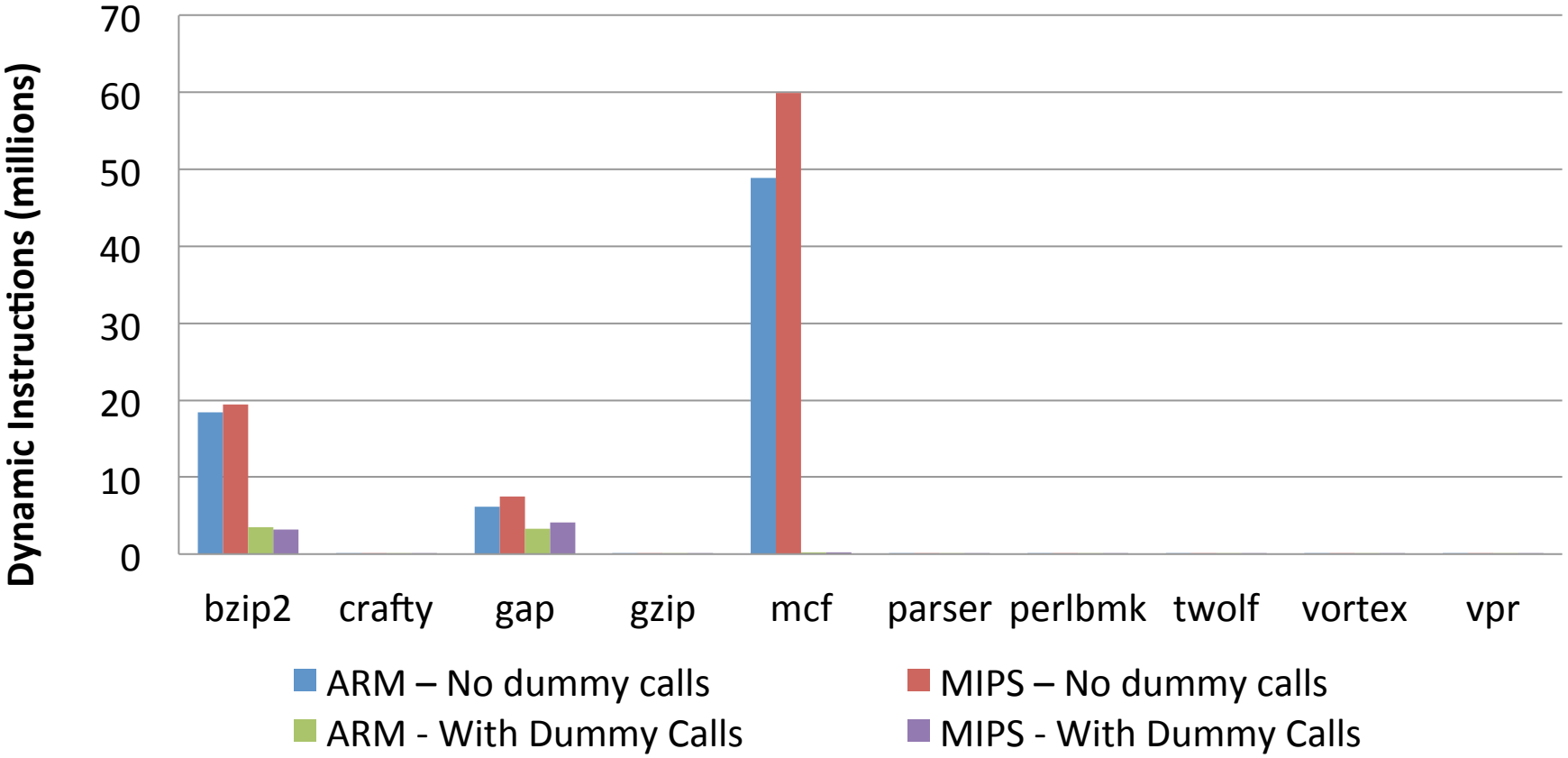
When the next function call is miles away…





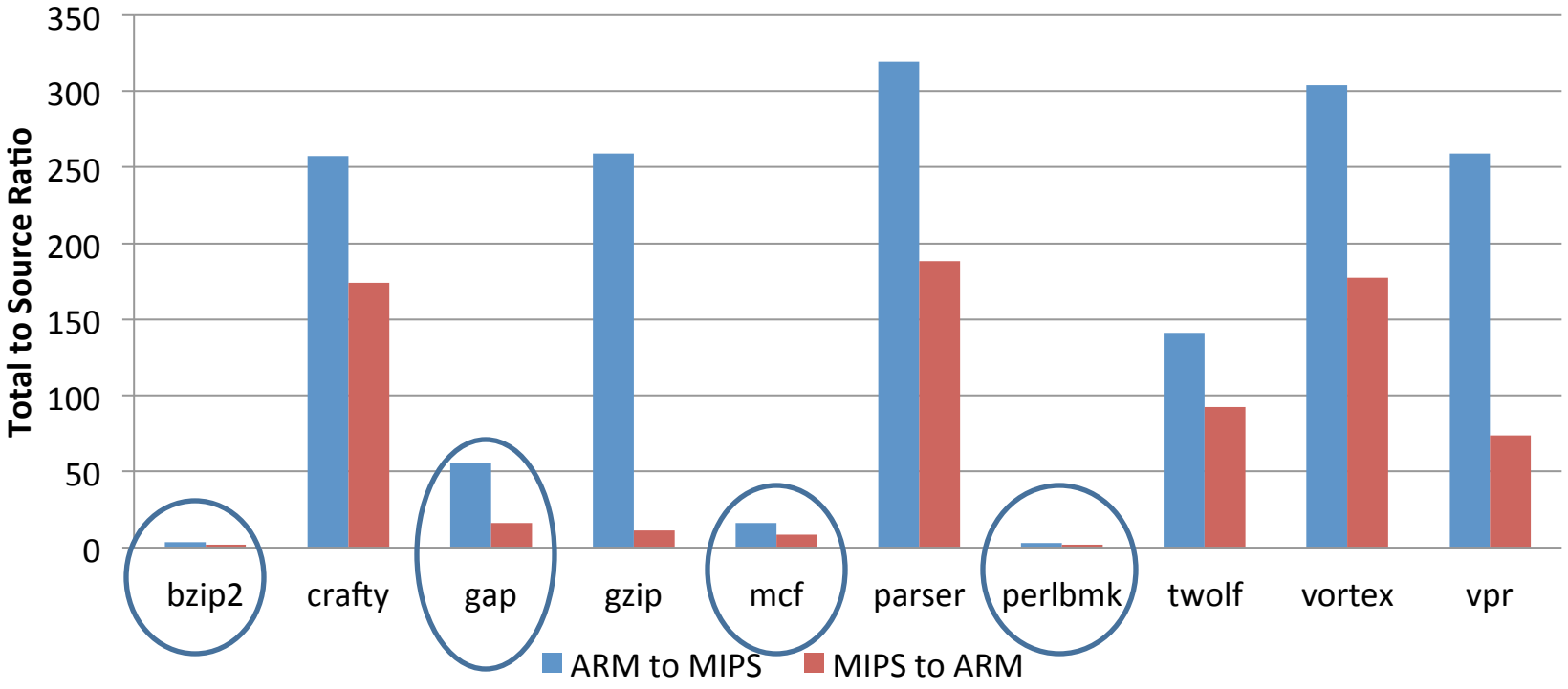
- Execution mostly happens from code cache by virtue of MEME chaining
- TOTAL/SRC ≈ TGT/SRC

Expected Time To Next Call Site



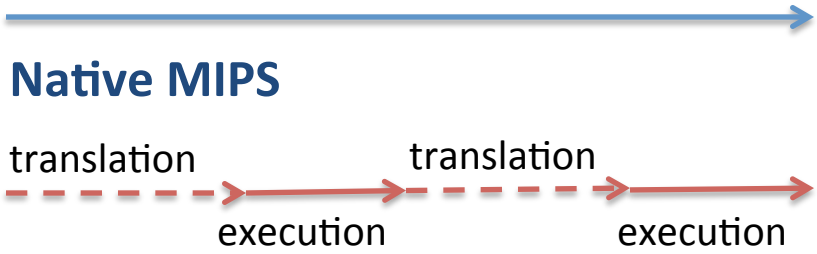
- Bzip, gap and mcf have millions of instructions to translate before next function call
- Dummy calls reduce binary translation time but affect native performance

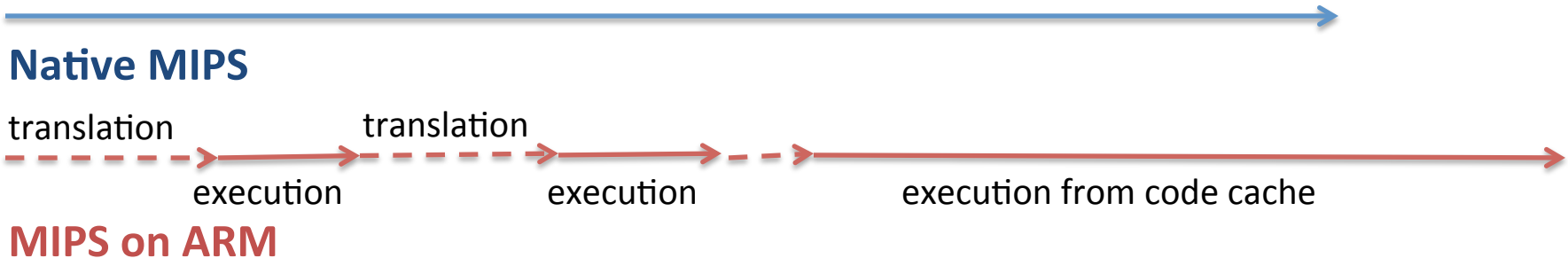
Binary Translation Costs
- Total to Source Ratio



- bzip2, gap and mcf have millions of instructions to translate before reaching a function call, and perl has a long running loop.
- They spend most of the time in code cache by virtue of MEME chaining

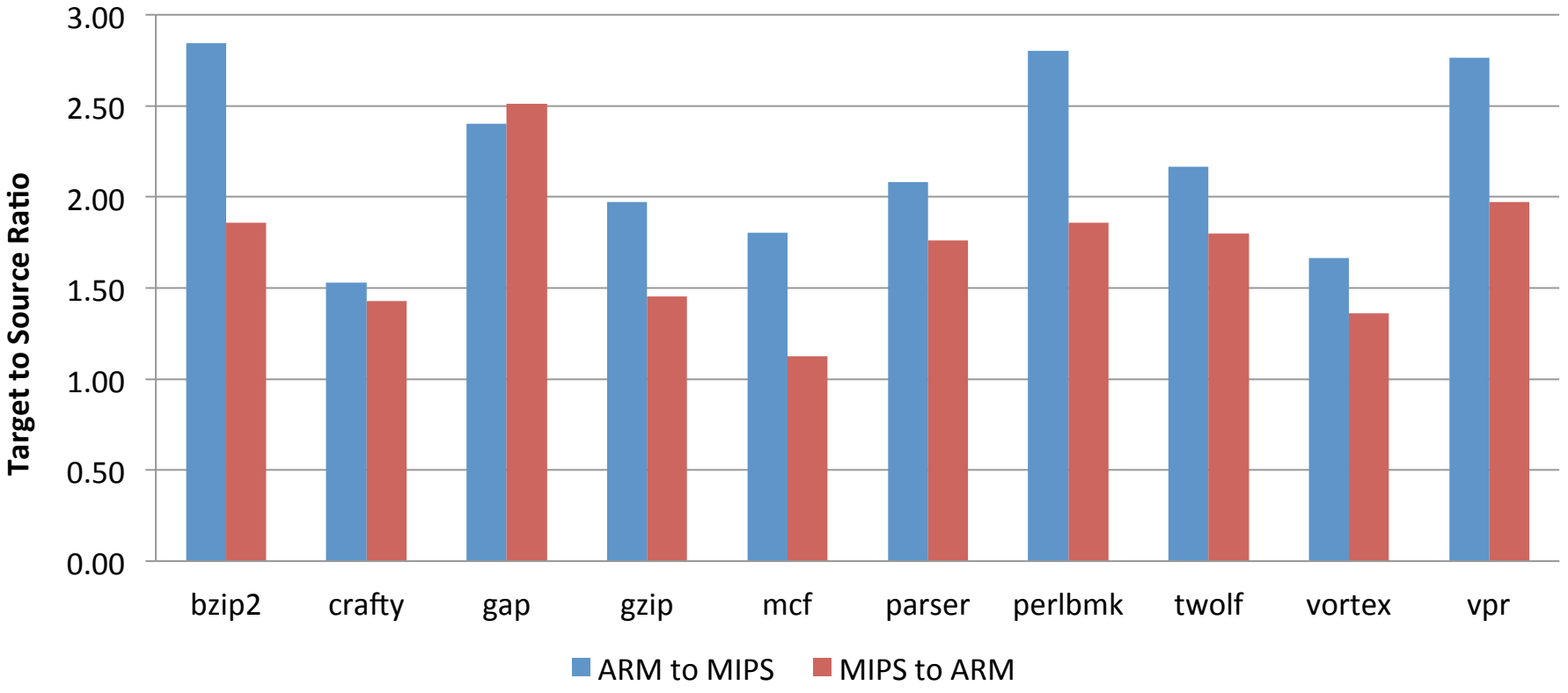
When the next function call is miles away…





- TOTAL/SRC ≈ TGT/SRC
- So, we still want TGT/SRC ratio to be low

Binary Translation Costs
- Target to Source Ratio

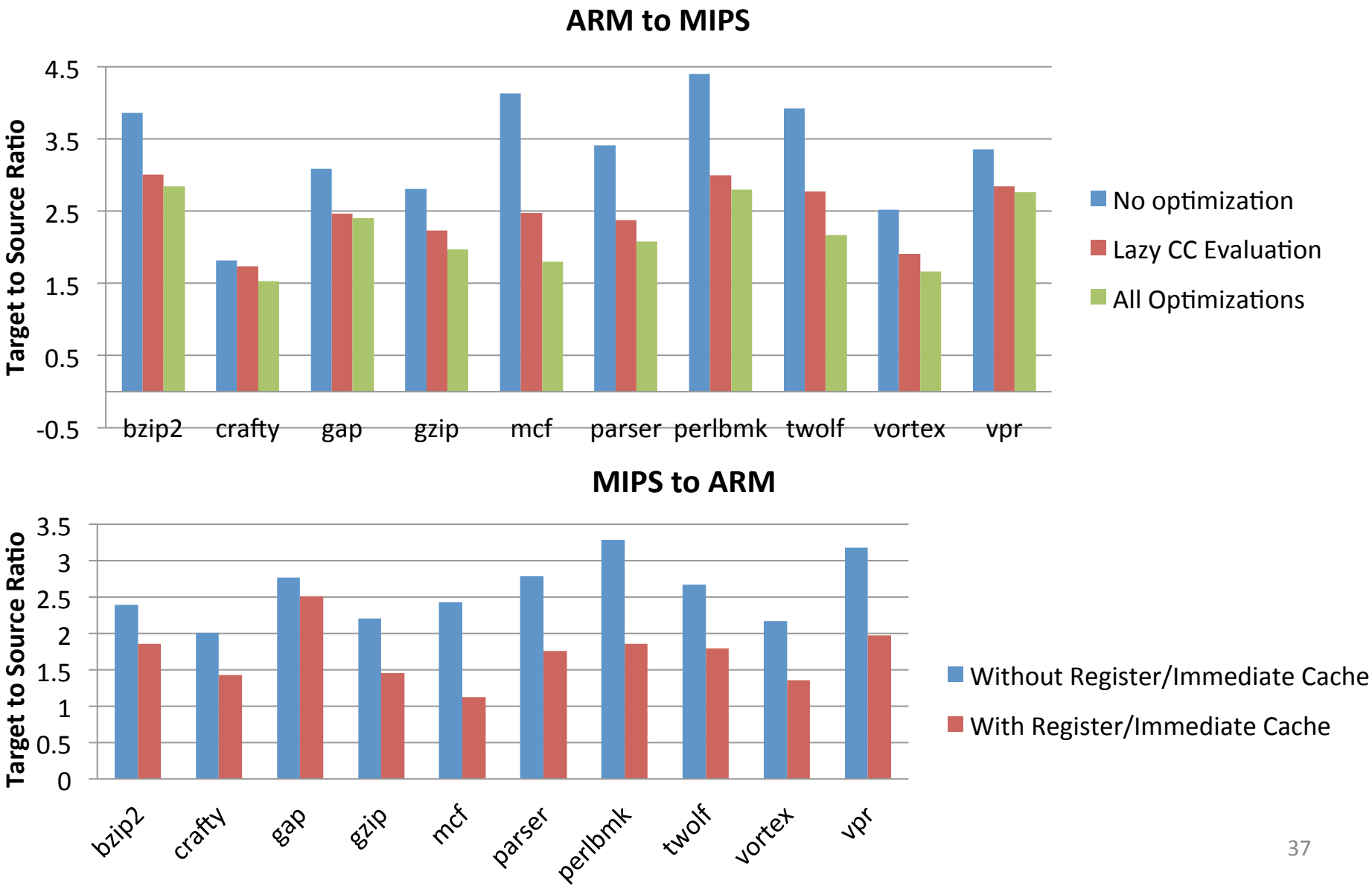


Lower the target-to-source ratio, better the performance

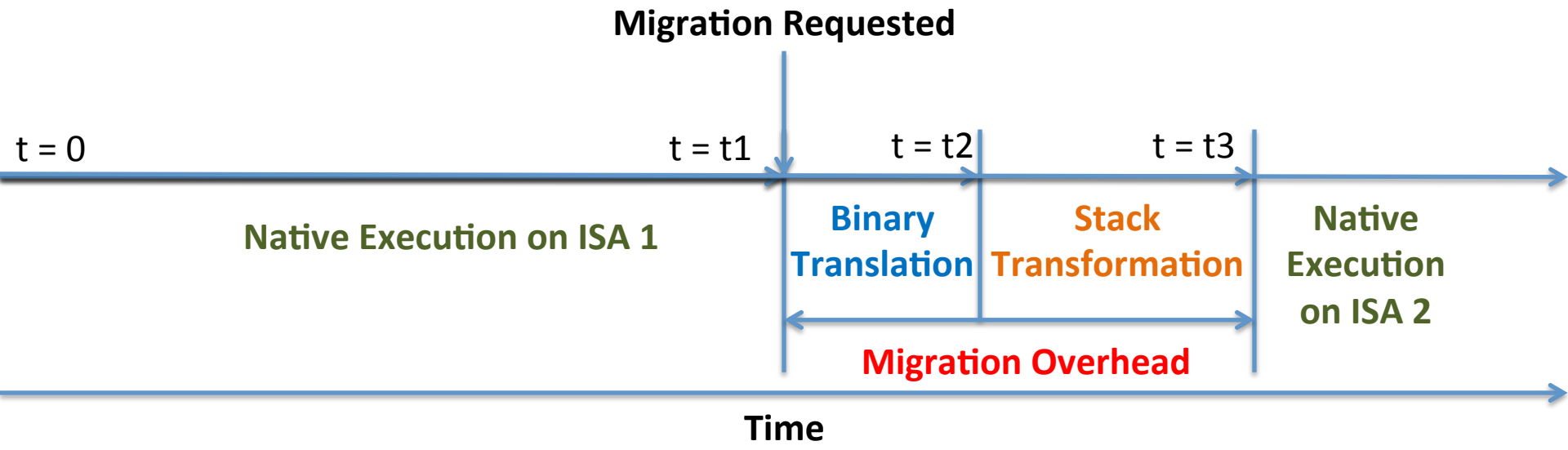
ISA-specific optimizations

- Lazy condition code evaluation
  - Evaluate a condition code only when necessary
- Register Allocation
  - Map frequently used SRC registers to registers in TGT
  - Cache frequently used unmapped registers
  - Use adaptive register allocation strategies
- Cache frequently used immediate values
- Other optimizations
  - Group predicated instructions
  - Lazy PC update
  - Constant folding/Constant Propagation

ISA specific optimizations



Execution Migration Timeline



Migration Overhead = Binary Translation + Stack Transformation

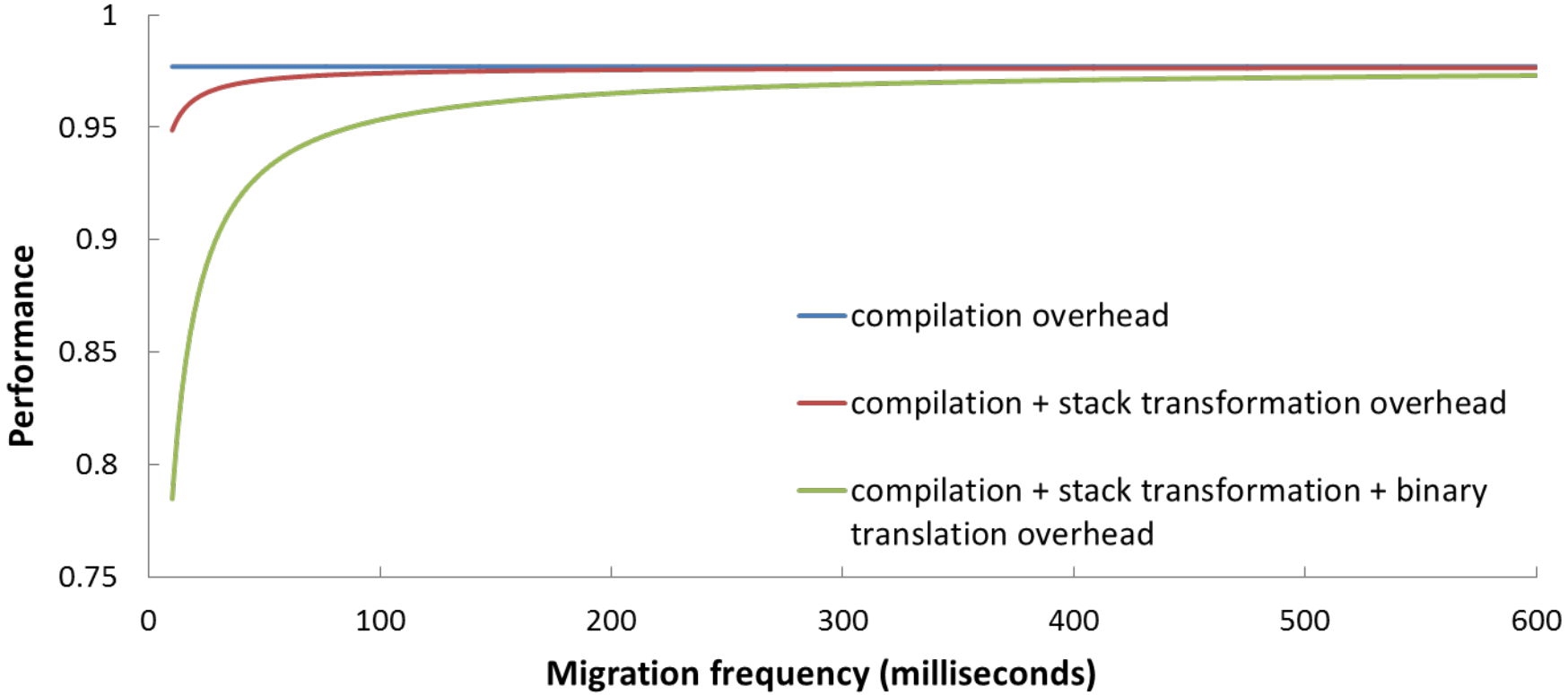
Total Performance Overhead =

Native execution (overhead due to static compilation) +

Binary Translation +

Stack Transformation

Total Performance Overhead



- Performance vs. migration frequency when migrating back and forth between an ARM core and a MIPS core
- With migrations occurring every 87 milliseconds (nearly every timer interrupt), performance drops down by just about 5%

Conclusion

- Current heterogeneous multicore architectures do not allow dynamic migration between heterogeneous cores.
- Recent research proposals allow migration, but constrain heterogeneity to a single ISA.
- Our execution migration strategy all but eliminates this barrier, enabling the architect to exploit the full benefits of heterogeneity.
- By significantly reducing the cost of memory transformation and employing fast binary translation, total overhead is reduced to less than 5% even if migrations happen at nearly every timer interrupt.

Thank You!