Register Relocation

Flexible Contexts for Multithreading

Carl A. Waldspurger

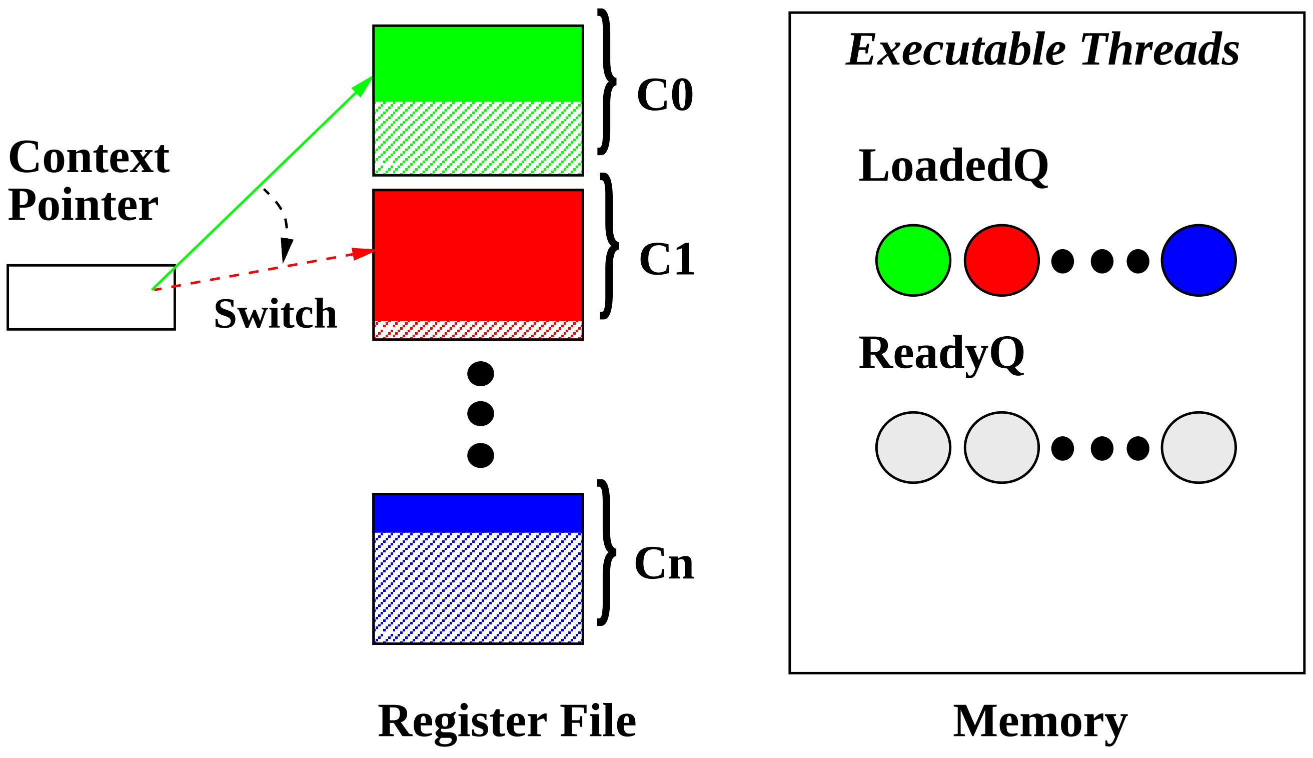
William E. Weihl

Parallel Software Group
MIT Laboratory for Computer Science

May 18, 1993

Multithreading

- **Goal: tolerate long latencies**
- **Approach: compute while waiting**
- **Mechanism: rapid context switching**



Flexible Contexts

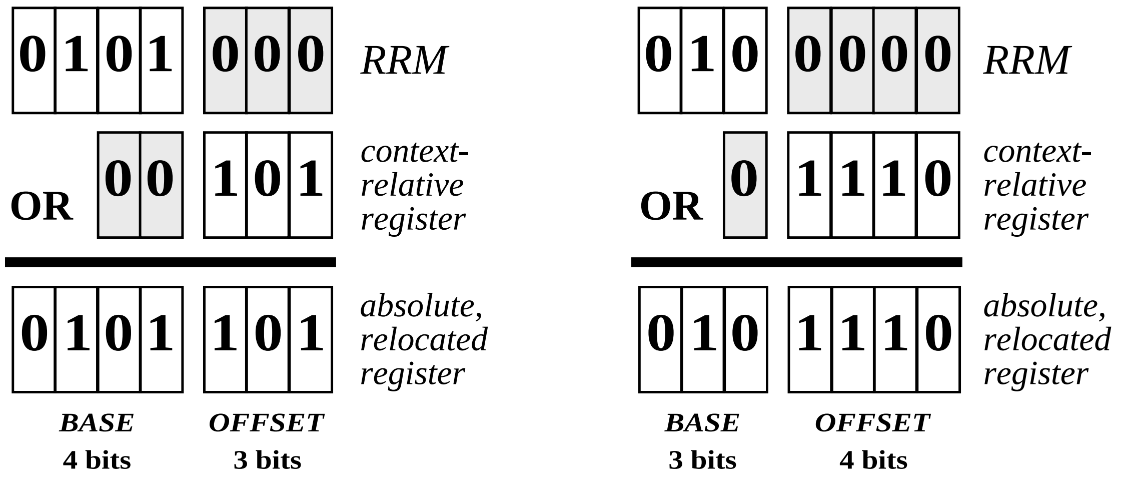
- **Thread requirements vary**
  - register usage is thread-dependent
  - decreasing marginal benefits from more registers
- **Software-based approach**
  - application-specific partitioning
  - variable-size contexts
  - static or dynamic division
- **More resident contexts**
  - better utilization of scarce registers
  - improve processor efficiency

Outline

- **Register relocation**
  - hardware primitive
  - software support
- **Experiments**
  - remote memory references
  - synchronization events
- **Related work**
- **Conclusions**
- **Future work**

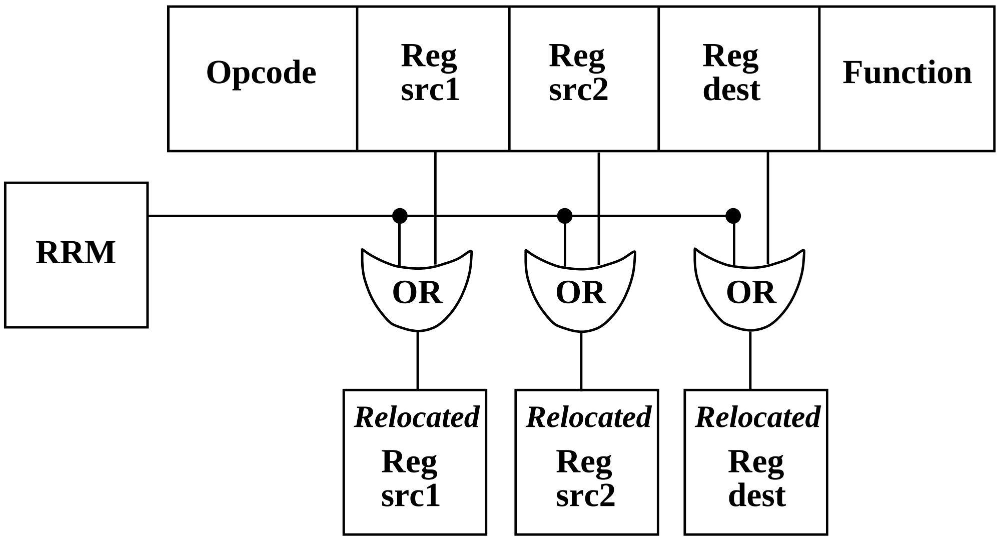
Register Relocation

- **Flexible base/offset scheme**
- **Base: register relocation mask (RRM)**
- **Offset: context-relative register numbers**
- ***Examples:***



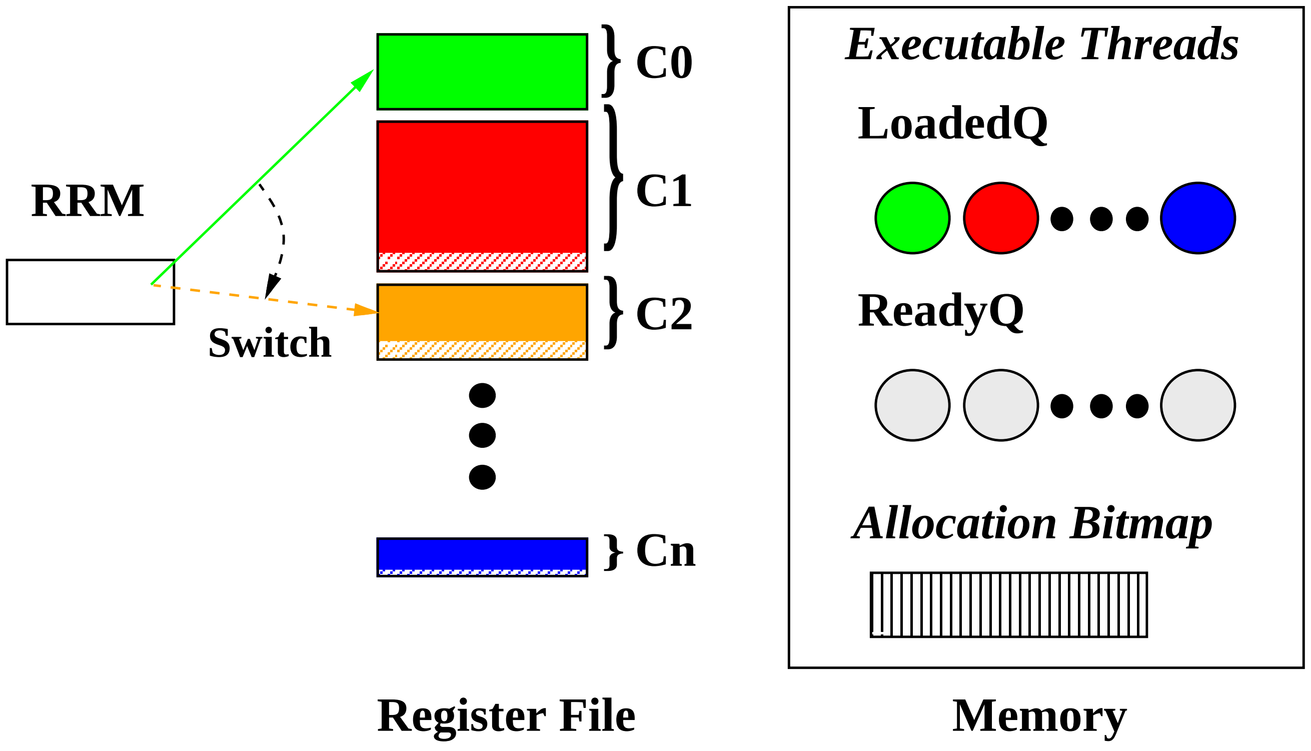
Hardware Support

- **Register relocation mask (RRM)**
  - special hardware register
  - $\lceil \lg n \rceil$ bits for *n* general registers
- **New instruction: ldrrm R**
  - set RRM from low-order bits of R
  - delay slots may follow
- **Instruction decode modifications**
  - bitwise OR instruction operands and RRM
  - RISC fixed-field decoding



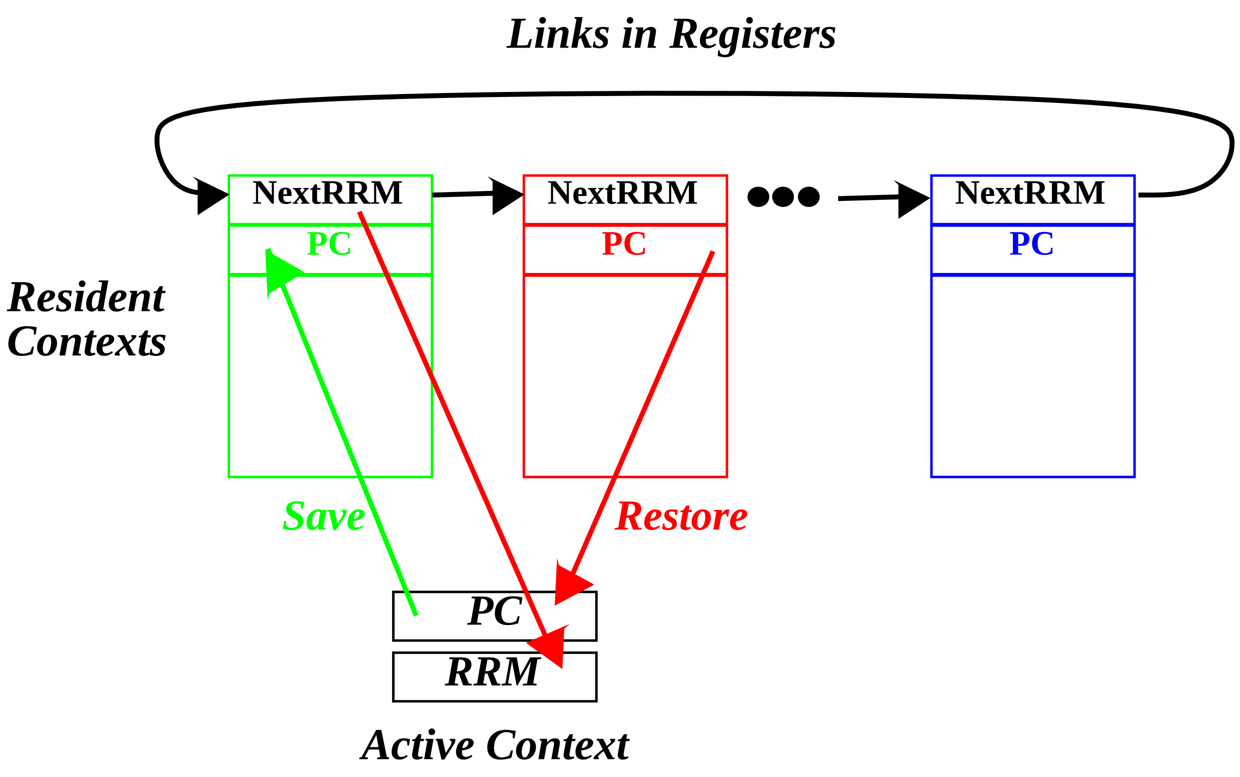
Software Support

- Context switch
- Context (de)allocate
- Context (un)load



Context Scheduling

- **Managed in software**
  - no hardware task queues
  - flexible control over policy
- **Sample policy**
  - resident context queue
  - round-robin scheduling
  - fast context switch ≈ 4 to 6 cycles



Context Management

- **Implemented in software**
  - flexible partitioning of register file
  - static or dynamic
  - identical or varying sizes
- **Context allocation**
  - general-purpose dynamic routines
  - search allocation bitmap
  - simple shift and mask operations
  - alloc $\approx$ 25 cycles, dealloc $\approx$ 5 cycles
- **Context loading**
  - save/restore exact number of registers
  - single routine with multiple entry points

Compiler Support

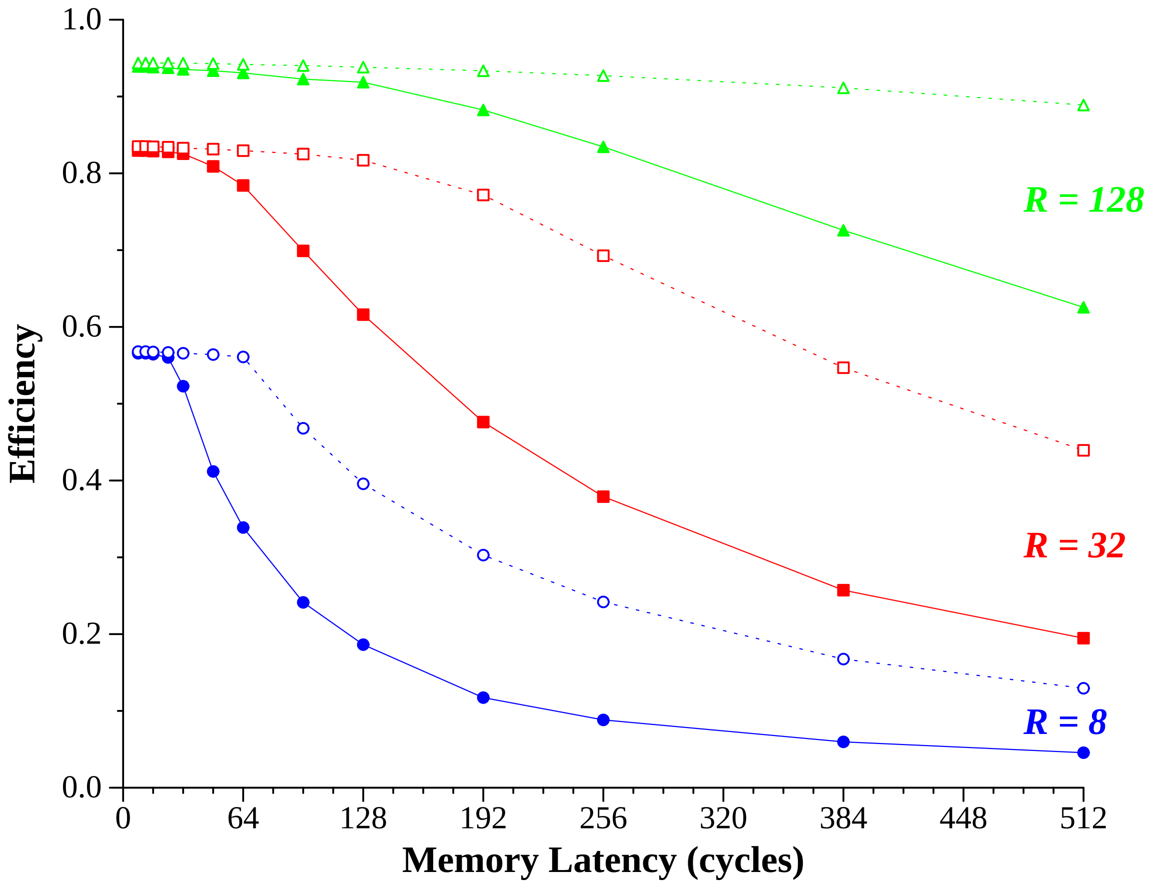
- **Compiler informs runtime system**
  - number of registers used by thread
  - computed by traversing thread call graph
- **Compiler protects thread contexts**
  - threads associated with single application
  - single address space
  - register and memory overwrites similar
- **Potential optimizations**
  - choose number of registers per context
  - decreasing marginal benefits
  - power-of-two context size constraint
  - *example:* allocate 16 vs. 17 registers

Experiments

- **Overview**
  - cache faults
  - synchronization faults
- **Conventional multithreading**
  - fixed-size contexts: 32 regs
  - zero alloc/dealloc costs
- **Register relocation**
  - variable-size contexts: 4, 8, 16, 32 regs
  - conservative alloc/dealloc costs
- **Simulation Environment**
  - single multiprocessor node
  - coarsely multithreaded architecture
  - synthetic threads with stochastic run lengths
  - Proteus simulator

Tolerating Cache Faults

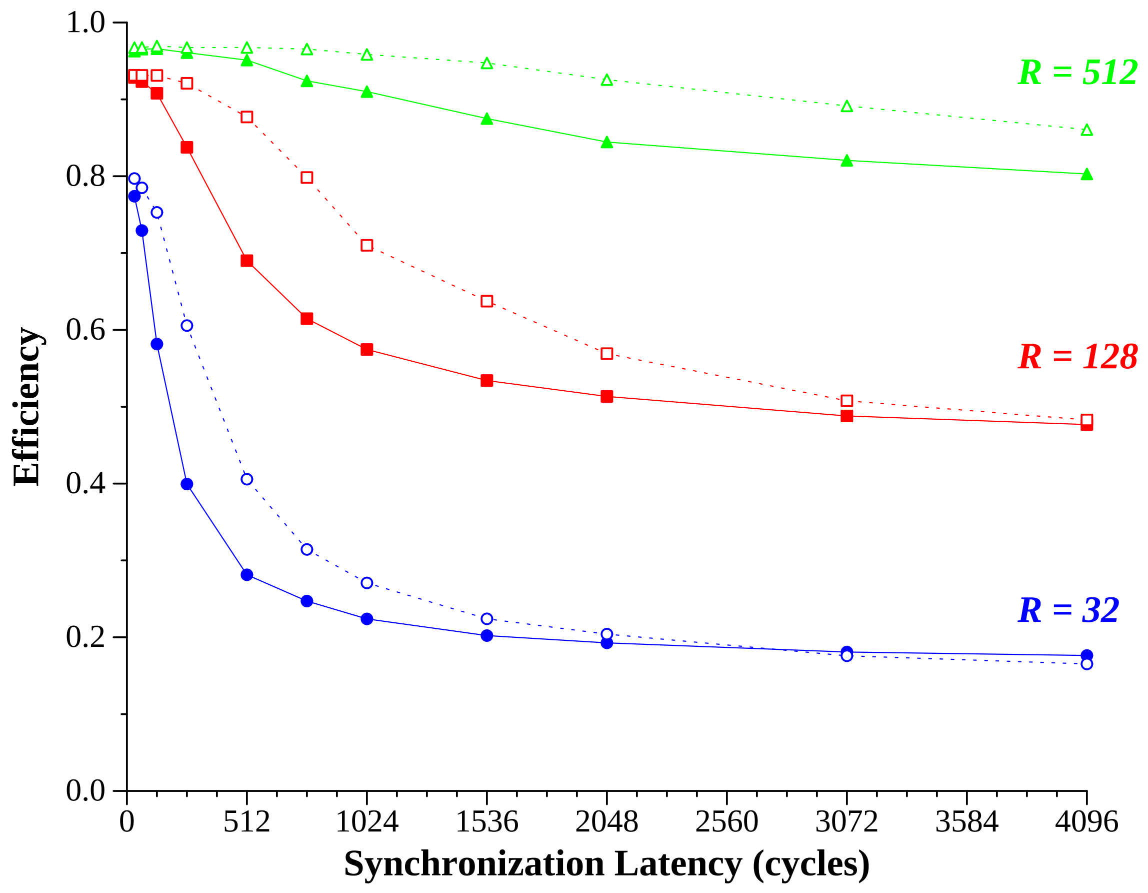
- **Parameters**
  - run lengths (R) geometrically distributed
  - remote memory latency constant
  - contexts never unloaded
- **Example results**
  - register file size = 128
  - threads require 6 to 24 registers



Tolerating Synchronization Faults

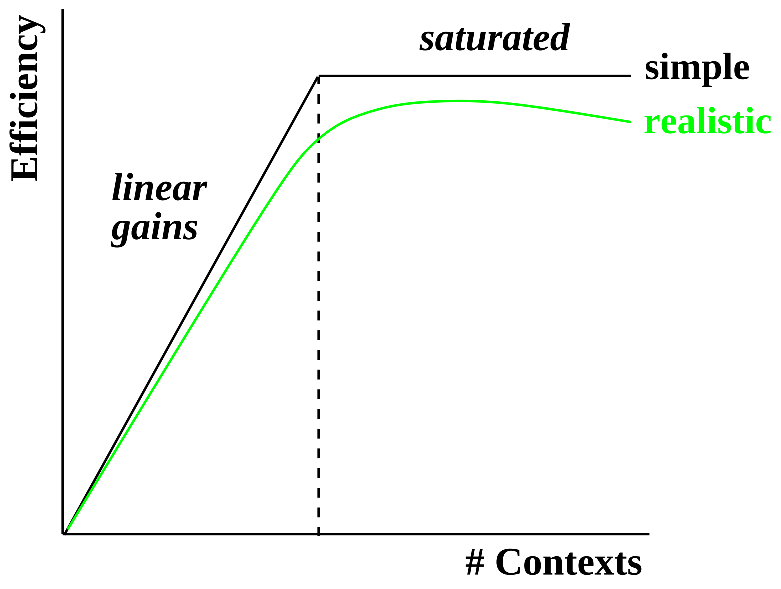
- **Parameters**
  - run lengths (R) geometrically distributed
  - synchronization latency exponentially distributed
  - competitive two-phase unloading policy

- **Example results**
  - register file size = 128
  - threads require 6 to 24 registers



Experiment Discussion

- **Many additional experiments**
  - similar results
  - both cache and synchronization faults
  - homogeneous context sizes
- **Significant performance improvements**
  - improved processor efficiency
  - better over wide range of parameters
  - 2× improvement for many workloads
- **Processor efficiency** [Saavedra-Barrera 90]



Related Work

- **Generally inflexible, hardware-intensive**
- **Finely multithreaded processors**
  - cycle-by-cycle interleaving
  - HEP, MASA, Horizon, Tera, Monsoon
- **Coarsely multithreaded processors**
  - execute longer instruction blocks
  - switch on high-latency operations
  - APRIL, hybrid dataflow/von-Neumann
- **Named State Processor**
  - fully associative register file
  - more flexible, but hardware-intensive
- **Base + offset register addressing**
  - addition flexible but expensive
  - Am29000, HEP

Conclusions

- **Register relocation**
  - multiple variable-size contexts
  - minimal hardware support
- **Significant flexibility**
  - software-based approach
  - flexible partitioning of register file
  - flexible control over scheduling
- **Substantial performance improvements**
  - better utilization of registers
  - enables more resident contexts
  - tolerate longer latencies, shorter run lengths
  - improved processor efficiency

Extensions and Future Work

- **Software-only approach**
  - generate multiple versions of code
  - use disjoint register subsets
- **Multiple active contexts**
  - select from multiple RRMs
  - context-specific operands
  - *example:* ADD C0.R3, C0.R4, C1.R6
- **Cache interference effects**
  - threads share common cache
  - most interference destructive
  - fine-grain parallelism shrinks working sets
  - utilization vs. interference tradeoff
- **Arbitrary context sizes**
  - addition vs. OR for relocation
  - efficient software support