Chapter 1

ENERGY-EXPOSED INSTRUCTION SETS

Krste Asanović, Mark Hampton, Ronny Krashinsky, Emmett Witchel
MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139, USA[*]
krste, mhampton, ronny, witchel@lcs.mit.edu

Abstract Modern *performance-oriented* ISAs, such as RISC and VLIW, only expose to software features that impact the critical path through computation. Pipelined microprocessor implementations hide most of the microarchitectural work performed in executing instructions. Therefore, there is no incentive to expose these micro-operations, and their energy consumption is hidden from software.

This work presents *energy-exposed* hardware-software interfaces to give software more fine-grain control over energy-consuming microarchitectural operations. We introduce *software restart markers* to make temporary processor state visible to software without complicating hardware exception management. This technique can enable a wide variety of energy optimizations. We implement *exposed bypass latches* which allow the compiler to eliminate register file traffic by directly targeting the processor bypass latches. Another technique, *tag-unchecked loads and stores*, allows software to access cache data without a hardware tag check when the compiler can guarantee an access will be to the same line as an earlier access.

Introduction

Power consumption is emerging as a key factor limiting computational performance in both mobile and tethered systems. Although there has been significant progress in low-power circuit design and low-power CAD and some work in low-power microarchitectures, there has been little work to date at the level of instruction set architecture (ISA) design for low power computing.

Modern ISAs such as RISC or VLIW are based on extensive research into the effects of instruction set design on performance, and provide a

[*]This work was partly funded by DARPA PAC/C award F30602-00-2-0562 and by NSF CAREER award CCR-0093354.

purely performance-oriented hardware-software interface. These instruction sets avoid providing alternate ways to perform the same task unless it will increase performance significantly. Implementations of these ISAs perform many energy-consuming microarchitectural operations during execution of each user level instruction and these dominate total power dissipation. For example, when executing an integer add instruction on a simple RISC processor only around 5% of the total energy consumption is due to the adder circuitry itself. The rest is dissipated by structures such as cache tag and data arrays, TLBs, register files, pipeline registers, and pipeline control logic. Modern machine pipelines have been refined to the point where most of the additional microarchitectural work is performed in a pipelined or parallel manner that does not affect the throughput or user-visible latency of a “simple” add instruction. Because their performance effects can be hidden, there is no incentive to expose these constituent micro-operations in a purely performance-oriented hardware-software interface — their energy consumption is hidden from software.

In this chapter, we present new energy-exposed hardware-software interfaces that give software fine-grain control over energy consumption. The key idea is to reward compile-time analysis with run-time energy savings. Instruction set enhancements enable this goal by providing software with alternative methods of executing an operation; performance is unchanged, but greater compile-time knowledge can be used to deactivate unnecessary portions of the machine microarchitecture. Our primary focus is on integer applications with complex control flow. This type of code will likely become the energy bottleneck in future embedded systems, as more regular computations can be mapped to energy-efficient vector instructions or custom hardware accelerators. We modify a RISC microprocessor architecture to support three energy-exposed techniques and develop compiler algorithms to target the enhanced instruction set.

The first technique is *software restart markers*, which reduce the energy expended in exception state management. Current pipelined machines invest significant energy in preserving precise exception semantics. Instruction results are buffered before being committed in order, requiring register rename logic to find the correct value for new instructions. Even a simple five-stage RISC pipeline has a bypass network that effectively performs these functions. In addition, other information such as PC and faulting memory addresses must be preserved in the pipeline until the exception can be serviced. Software restart markers reduce energy by allowing the compiler to annotate at which points it requires precise exception behavior. Initial results show that the number of precise exception points can be reduced by a factor of three. More importantly, this technique allows additional machine state to be made visible between restart points, enabling the introduction of more energy-exposed features without incurring additional exception management costs.

The second technique is *exposing bypass latches* with a hybrid RISC-accumulator architecture that uses compile-time register lifetime information to reduce the number of register file reads and writes performed at run time. Many register values in a computation are short-lived, being produced by one instruction, consumed by the next instruction, and then never used again. This register lifetime information can be encoded by adding accumulator registers to a general-purpose register (GPR) RISC architecture, which allow software to pass values directly from one instruction to the next without accessing the GPRs. The accumulator registers can be mapped to the bypass latches that are already present in a CPU datapath. To avoid the hardware and energy overhead which would be necessary to preserve this bypass latch state during exceptions, the bypass latch accumulator registers are treated as temporary state which is recreated when a region is re-executed after any trap. This technique can remove a third of all register file writes.

The final technique is *tag-unchecked loads and stores*. Tag accesses consume over half the energy of a data cache access in a low-power microprocessor. In cases where compile-time analysis can guarantee that two accesses will be to the same cache line, tag-unchecked loads and stores allow the hardware to avoid performing a tag-check on the second access. Initial results indicate up to 70% of tag checks in SPECint95 and Mediabench programs can be removed at compile time.

1. Baseline Processor

For mobile and embedded processors, both energy and performance are of interest, and a traditional pipelined datapath is preferable to a more complex superscalar design [8]. An energy-efficient five-stage pipelined MIPS RISC microprocessor based on that presented in [4] was adopted as a baseline with which to evaluate the three energy-exposed instruction set techniques. The basic pipeline structure of this design is shown in Figure 1.1. The design has split 16 KB instruction and data caches that are organized as 64-way set-associative CAM-tag caches with 32-byte lines. This pipeline and cache configuration is designed to be similar to the popular StrongARM-1 [6] low-power microprocessor.

2. Software Restart Regions

Machines that support an operating system with preemptive context switching or demand-paged virtual memory must provide some mechanism to manage exceptions. If precise exceptions are supported in a pipelined machine, hardware must either buffer state updates in some form of future file until all

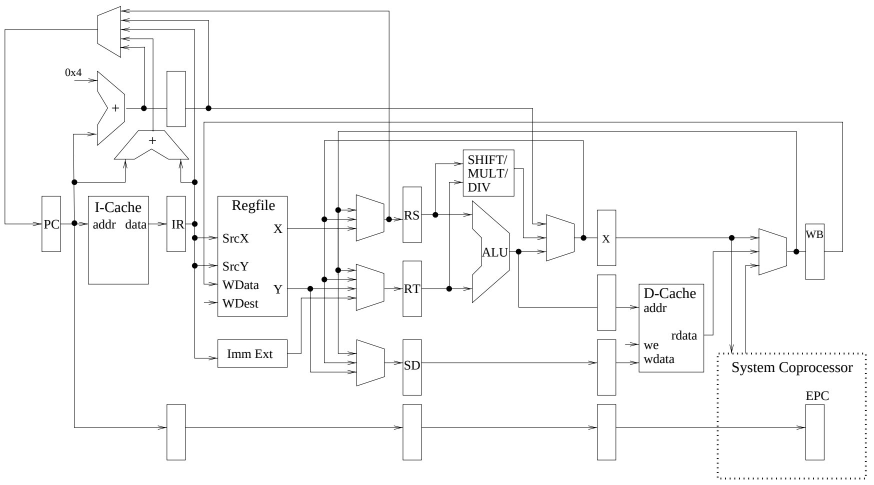


Figure 1.1. Baseline pipeline design.

possible exceptions have cleared, or alternatively save old machine state in history buffers so that it can be recalled when an exception is detected [9]. Even if precise exceptions are not implemented, hardware must provide access paths to allow software to save and restore all machine pipeline state across exceptions to allow restart [11]. Note that these schemes add additional exception state management energy overhead to the execution of all instructions. An energy-exposed instruction set requires that internal machine state is made visible to software. However, if making this additional machine state visible to software incurs additional exception management overhead, much of the potential energy savings could be lost.

Most current ISAs have sequential instruction semantics and implementations are usually required to provide precise exceptions. That is, exceptions must be taken in program order, and on recognizing an exception the machine must provide the program counter (PC) of the faulting instruction and ensure that all earlier instructions have committed state updates and no later instructions have affected architectural state [9]. This provides a simple model for user code, which is given the illusion of uninterrupted execution; and for system software, which only need save and restore programmer-visible state including the faulting PC to swap contexts. However, supporting these semantics incurs hardware exception management overhead for every instruction executed. In practice, exceptions occur rarely and usually do not require full precision in exception reporting. For example, timer interrupts and page faults only require that the process be restartable.

Software restart markers reduce the energy cost of exception management by requiring software to explicitly divide the instruction stream into restartable regions. After handling a trap, the OS resumes execution at the beginning of the restart region for the associated instruction. A conventional architecture with precise exceptions is equivalent to placing every instruction into its own restart region. A simple local analysis can remove many of these implicit restart points by placing multiple instructions into one region.

2.1 Restart Marker Implementation

Restart points are encoded by marking the last instruction in a restart region. This instruction is called the *barrier* instruction because it acts as a trap barrier that will commit and irrevocably update machine state only if it is guaranteed that it will not raise an exception and that any preceding instruction will not raise an exception. Also, the barrier instruction ensures that if an exception does occur before it commits, the effects of following instructions will not be visible. When the barrier instruction commits it will update a kernel visible register, the *restart program counter*, to point to the next instruction to be executed; this instruction is the beginning of the next restart region. Note that marking every instruction as a barrier instruction is equivalent to conventional precise exception semantics.

The compiler must ensure that the code in a region is such that the operating system kernel can restart the process after an exception by simply jumping to the restart PC. This requires that software ensure the code in each restart region (except for the final barrier instruction) is idempotent, i.e., that it can be re-executed multiple times without changing the result. This restriction still allows for the creation of large restart regions. For example, many of the functions in the standard C library can each be entirely contained within one restart region. To illustrate this point, consider the `sprintf` function, for which the prototype is given below.

```
int sprintf(char *s, const char *format, ...);
```

This function uses the `format` string as an input argument to write to the string pointed to by `s`. As long as the input argument to the function is passed in stack memory and not altered by the routine (meaning that the input arguments and output arguments cannot overlap in memory), the function can be restarted multiple times and still produce the same result. For the `sprintf` function, the `format` argument is declared to be `const`, so it is not modified. Thus, the function satisfies the criteria for idempotency. In general, any function that does not modify its arguments can use an arbitrary amount of local read/write workspace and still be idempotent.

2.2 Categories of Machine State

The use of software restart markers introduces three different types of machine state: checkpointed, stable, and temporary. Checkpointed state is copied into checkpoint registers each time a barrier instruction commits, and is recoverable if an exception occurs. In the simple scheme used in this chapter, the only checkpointed state is the restart PC. Stable state is preserved across an exception by the kernel: conventional registers and memory fall into this category. Finally, temporary state is only valid within a restart region, and is not preserved across an exception. The bypass latches discussed in the next section are an example of temporary state.

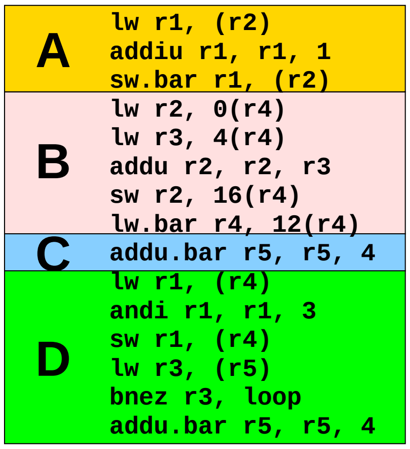
The power of software restart markers is that a great deal of internal machine pipeline state can be exposed by mapping it to temporary state without needing hardware support to preserve this state across exceptions. In fact, the temporary state can be ignored by the operating system. As long as the kernel saves the visible stable user state and the restart PC, the process can be restarted after each exception.

It is interesting to note that the MIPS instruction set [3] already incorporates a limited form of restart regions to support delayed branches, i.e., the next PC is user-visible temporary state that is updated by branch instructions but which is never saved and restored by the kernel. Branches are always idempotent and never trap barriers so that any trap on a delay slot instruction restarts at the branch itself to recreate the next PC.

2.3 Example Use

Figure 1.2 presents some example code showing how the restart barriers are used. The code is similar to conventional MIPS code, except that barrier instructions are marked with a `.bar` suffix. The barrier instructions split the code into four regions labeled A, B, C, D. If an exception occurs within a region, the code can be restarted from the beginning of the region. For example, if the store in region A encounters a write fault (perhaps because the page was write protected as part of a copy-on-write protocol), the OS can replay the code starting at the initial load and obtain the same result. The store in region A must be marked as a barrier because its update of memory would otherwise cause the region to be non-idempotent.

Region B contains a store that is not marked as a barrier. If the final load in region B takes a page fault, this store will have likely changed memory irrevocably. Nevertheless, the OS can restart region B from the initial load instruction because the region is idempotent. The final load is marked as the barrier because it overwrites the original value of the pointer `r4` which would be needed to restart the region.



```
lw r1, (r2)
addiu r1, r1, 1
sw.bar r1, (r2)
lw r2, 0(r4)
lw r3, 4(r4)
addu r2, r2, r3
sw r2, 16(r4)
lw.bar r4, 12(r4)
addu.bar r5, r5, 4
lw r1, (r4)
andi r1, r1, 3
sw r1, (r4)
lw r3, (r5)
bnez r3, loop
addu.bar r5, r5, 4
```

Figure 1.2. Code example showing restart regions. Instructions with a `.bar` suffix are the barrier instructions at the end of each region.

Region C contains a single instruction, and shows how a simple compiler can fall back to marking all instructions as barrier instructions to replicate conventional precise exception semantics.

One sufficient, but not necessary, condition for idempotency is that the set of all external sources (registers and memory) read by the region is disjoint from the set of destinations written by the region (note that it is acceptable to overwrite a value produced within the region). This is not a necessary condition as shown by the example in region D. Here, the store to memory changes input source data, but with an idempotent operation (masking out the bottom two bits). The barrier for this region is placed on the delay slot of the branch instruction which will record the branch target in the restart PC when it commits.

2.4 Compiler Analysis

Restart analysis was implemented as a pass within the assembler that performs a purely local optimization at the basic block level after instruction scheduling and register allocation. The analysis begins a new restart region at the start of a basic block, then scans the sequence of instructions, updating the set of external values read and the internal set of values written. When a conflict is detected, the conflicting instruction is marked as a barrier instruction, then the read and write sets are cleared and a new region is started. Barriers are also placed in front of system calls and any other instructions that will likely cause a trap. Although we use a single bit in each set to represent memory, we also incorporate a limited form of memory analysis by separately tracking the base register and offset for each memory instruction. When the first memory instruction in a restart region is encountered, we store the base register, and we create a linked list to hold the offset. For each subsequent memory instruction, if the same base register is used, we can look at the offset to determine whether the access is to a distinct memory location from all previous memory operations. If the base register is modified, or a different base register is used, we revert to treating all of memory as a single location.

One concern with the restart scheme is ensuring forward progress in the face of finite resources. If TLB misses are handled by software exception handlers, then the number of memory operations in a region must be restricted to be less than the number of available TLB entries to ensure that the region can run from start to finish without incurring a TLB fault. Similarly for demand-paging, the number of physical pages must be greater than the number of memory operations allowed in a region. These restrictions can be enforced by the compiler and checked by the operating system, which can abort a process if it fails to progress through a region.

2.5 Evaluation

The results of this restart analysis are shown in Figure 1.3 for SPECint95 and Mediabench benchmarks. The Figure shows the number of dynamic instructions that are restart points for both baseline MIPS code and for code after the restart analysis. For baseline MIPS code, only branches and jumps do not have barriers and so around 79–95% of all instructions have barriers. After the simple local restart analysis, only 25–40% of instructions are barriers with an average of around 3 instructions in each restart region. More aggressive compiler analysis should generate even larger regions, and allow entire functions to be placed into a single restart region.

For the simple five-stage pipeline, restart analysis by itself only results in a minor energy saving in the exception PC pipeline. The instruction pipeline tags each instruction with its PC as it moves down the pipeline to identify the faulting instruction on an exception. The PC is latched into the EPC register in the system coprocessor if an exception occurs (Figure 1.1). With the restart analysis only the barrier instructions cause an exception PC to shift down the pipeline, allowing the PC pipeline to be gated off in other cases.

The primary advantage of software restart markers is that they make it possible to expose the internal details of a processor to the compiler as temporary state in between restart points. The next section illustrates one use of temporary state to save energy for register file traffic.

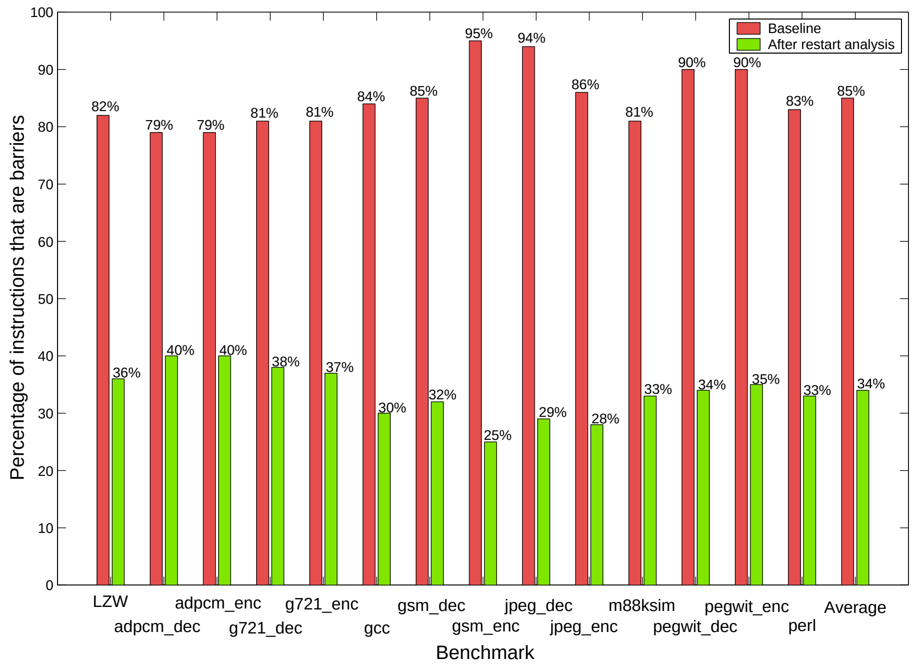


Figure 1.3. Percentage of dynamic barrier instructions for baseline MIPS code and code after restart analysis. For baseline MIPS, only branch and jump instructions exclude barriers.

3. Exposing Bypass Latches with a Hybrid RISC-Accumulator Architecture

Simulations of the MediaBench and SPECint95 benchmarks reveal that around half of the values written to the register file are used exactly once, usually by the instruction executed immediately after the one producing the value. For example, in the code sequence to increment a memory variable,

```
lw  r1, (r3)    # Load value.
add r1,  r1, 1  # Increment.
sw  r1, (r3)    # Update memory.
```

the result of the load and add are only used once by the subsequent instruction and are normally read from the bypass network rather than the register file. A conventional processor must assume that the register could be read at any arbitrary point in the future or that there could be an exception right after the instruction and hence must always write the value into the register file. The compiler must already have calculated register lifetime information to perform register allocation but has no convenient mechanism to communicate this information to hardware through a standard instruction set.

3.1 ISA Enhancements

By giving software explicit control of the bypass latches, it is possible to reduce the register file traffic considerably. For example, the above code can be rewritten as:

```
lw     RS, (r3)     # Load RS latch.
add    SD,  RS, 1   # Increment, put result in SD.
sw.bar SD, (r3)     # Update with barrier.
```

where the `RS` operand specifies the use of the bypass latch in front of one input to the ALU and the `SD` operand specifies the use of the bypass latch that holds data being stored to memory (Figure 1.1). Note that this sequence has the same performance as the previous sequence but now two writes and two reads of the register file have been avoided and replaced with accesses to the bypass latches. In effect, when using the bypass latches, software turns off the register fetch and write back stages of the machine pipeline, and thereby removes microarchitectural energy overhead.

The final store is marked as a barrier instruction, because it performs a non-idempotent memory update. If an exception is taken on any instruction in the sequence, the code can be restarted from the load instruction. The bypass latch does not have to be saved and restored by the operating system because the bypass latch state will be recreated when this region is restarted.

This modification creates a hybrid RISC-accumulator architecture, but without the need to preserve accumulator contents around exceptions. This allows the accumulator registers to map directly to the bypass latches that were present in the original pipeline design, with no area, delay, or energy penalty for additional backup state or access paths to support exceptions.

Exposing bypass latches can eliminate register file reads when a temporary value is obtained from a bypass latch instead of the register file. Additionally, register file reads can be eliminated by another method which is referred to as *read caching* [10]. For example, when a procedure is called, it may save registers on the stack before using them, as shown in the following code segment:

```
sw r3, 8(sp)
sw r2, 4(sp)
sw r1, 0(sp)
```

The value in the stack pointer register does not change in the above sequence, yet it is read from the register file and clocked into the `RS` bypass latch for each instruction. Software can eliminate these extraneous reads as well as the unnecessary clocking of the `RS` latch by the use of explicit bypass latches, as shown in the following rewritten code segment:

```
sw r3, 8(sp)
sw r2, 4(RS)
sw r1, 0(RS)
```

3.2 Compiler Analysis

Our implementation of the exposed bypass latch code takes advantage of the static liveness information that is already maintained by the compiler. When the compiler determines that a value read by an instruction is being referenced for the last time—i.e. the value will be dead after the instruction executes—it appends a "`.l`" suffix to the assembly opcode with a corresponding operand number to indicate the last use of the value.

The liveness information generated for each instruction is then used by the scheduler that we added to the assembler. The scheduler reorders instructions within a basic block. It performs several passes on the code. First, it attempts to maximize performance by reordering instructions to mask latencies that can cause pipeline stalls–in particular, it tries to fill load-use delay slots with independent instructions. It also attempts to fill the architected branch delay slot. Next, the scheduler uses the lifetime information generated by the compiler to determine if bypass latches can be used in place of general-purpose registers to statically bypass a value. In the subsequent pass, the scheduler creates the restart regions discussed in the previous section. It then looks for read caching opportunities, and finally tries to perform additional static bypassing from the memory stage of the pipeline.

Note that static bypassing from the memory stage raises additional constraints not required for bypassing from one instruction to a subsequent instruction. Consider the following example:

```
add r1, r2, r3
sub r4, r5, r6
and r7, r1, r4
```

In the above code segment, `r1` is read for the last time by the `and` instruction. This would appear to provide an opportunity for static bypassing from the memory stage by having the first `add` instruction target the X latch (Figure 1.1). However, in this scenario, if there is an instruction cache miss for the `and` instruction, the `sub` instruction will overwrite the value in the X latch as it proceeds through the pipeline. To avoid this problem, we must either require strict pipeline sequencing, so that instructions go down the pipeline together with no bubbles between them, or we must not permit an instruction which overwrites the X latch (e.g., the `sub` instruction in the above example) to be the intermediate instruction in a memory stage bypassing sequence. We chose the latter option, as this placed no additional constraints on the hardware implementation. Since they do not write back to the register file, instructions which target the bypass latches are candidates for the intermediate instruction in a memory stage bypassing sequence, for example:

```
add  X, r2, r3
sub RS, r5, r6
and r7,  X, RS
```

3.3 Evaluation

For our simulations, we modeled the RS, RT, SD, and X bypass latches by reserving four general-purpose registers in the compiler and using their specifiers in the scheduler when modifying an instruction to target a bypass latch. We observed that the loss of these registers in the compiler's register allocator did not have an adverse effect on performance. Ideally, the instruction set encoding would be designed to support bypass latches directly.

The reduction in register file writes is shown in Figure 1.4. On average, 34% of all writes are eliminated. The reduction in register file reads is shown in Figure 1.5. On average, 28% of all reads are eliminated.

4. Tag-Unchecked Loads and Stores with Direct Addressing

The memory system, including caches, consumes a significant fraction of total system power. One significant source of energy consumption is the tag check in the primary data cache. *Direct addressing* allows software to access

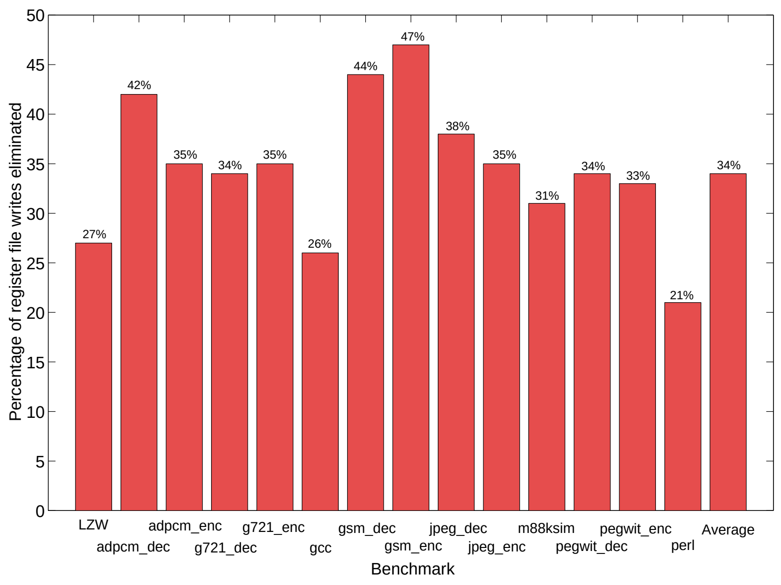


Figure 1.4. Percentage of register file writes eliminated in energy-exposed processor.

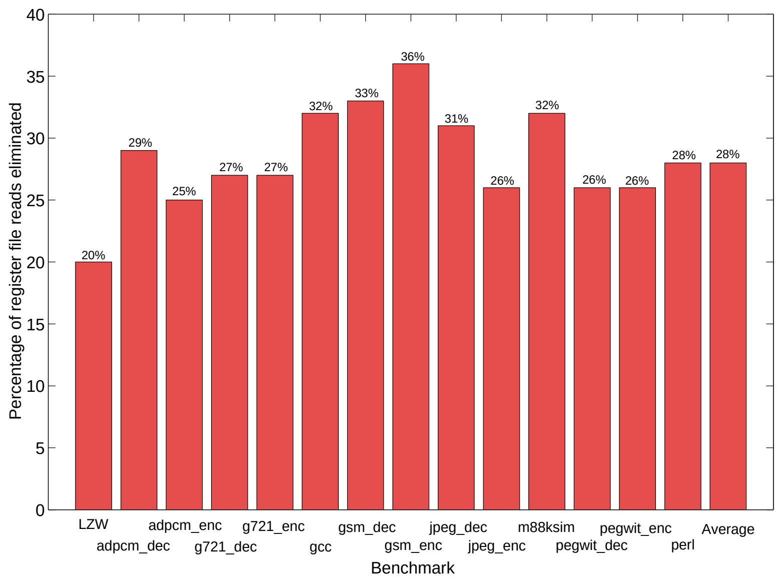


Figure 1.5. Percentage of register file reads avoided in energy-exposed processor.

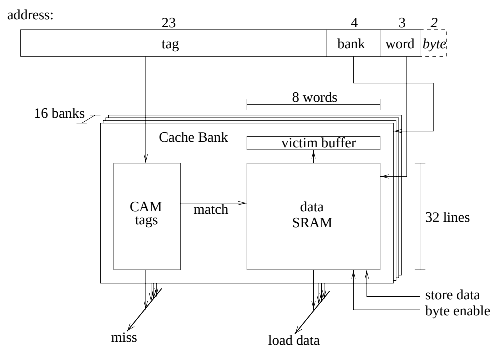


Figure 1.6. The organization of a highly-associative CAM-tag cache. The tag bits are broadcast to the CAM, and if there is a hit, the word is read out.

cache data without the hardware performing a cache tag check. These *tag-unchecked loads and stores* save the energy of performing a tag check when the compiler can guarantee an access will be to the same line as an earlier access. If the compiler cannot determine this information, or if cache lines are evicted due to interrupts or cache invalidations, direct addressing gracefully degrades back to conventional tag-checked accesses. Direct addressing is only used for data accesses since instruction caches, while they dissipate considerable energy, have very regular access patterns and are only accessed via the program counter. Hence they are amenable to software-invisible micro-architectural techniques for power reduction.

Commercial low-power processors usually employ highly associative CAM-tag caches [1, 7, 6, 2]. Direct-mapped caches, although simpler, are inefficient in terms of energy usage because they experience more misses due to conflicts. Figure 1.6 shows the organization of a CAM-tag cache. Although CAM-tag caches reduce miss rates and hence total access energy, they expend relatively greater energy in tag checks [13]. The tag check for CAM-tag caches is expensive because the tag is broadcast to the CAM in order to find the proper line for the data. If we could shortcut that process—if the software could tell the hardware what line to read, rather than providing a virtual address as a key to the content-addressable memory—then we would save significant amounts of energy. The problem is how to let software directly access cache lines without compromising inter-process protection and while preserving correct operation in the face of cache replacements or other cache coherence actions.

4.1 ISA Enhancements

To eliminate tag checks, the processor-cache interface is enhanced with a direct access mode that tells the hardware exactly where the data is located. The processor state is augmented with some number of *direct address* (DA)

Instruction	*Explanation*
`(l|s)wlda rt, off(rs), da`	Load or store word, load direct address. These instructions act like regular loads and stores, but they also set the direct address register `da` with the location of the referenced line. We use MIPS as the basis of our instruction encoding, so the offset for this instruction is 13 bits signed instead of the regular 16-bit offset since there are 3 bits used as a `da` specifier.
`(l|s)wda rt, off(rs), da`	Load or store word, using direct address. Cache data from the line pointed to by `da`, using the line offset bits of `rs + off` is transferred to register `rt` (or the contents of `rt` is stored into the line specified by `da`). If `da` is invalid, the instruction acts like `(l|s)wlda`, accessing memory and setting the `da` register.
`jr.dainv rs, mask`	Jump register and invalidate direct address registers. This acts like a jump register instruction, but it also clears the valid bit on the direct address registers specified in the (little endian) bitmask. This instruction is used at the end of function calls and by the operating system when the DA register lifetime has ended.

Table 1.1. A table of instructions for manipulating and using direct address registers

registers. These registers contain enough information to specify the exact location of a cache line in the cache data RAM, and also have a valid bit. The exact width and data layout of the DA register is invisible to software to avoid exposing the implementation-dependent structure of the cache. In particular, software is only made aware of the length of a cache line, but not the total cache capacity or associativity.

Table 1.1 shows the instructions needed by the CPU to use the DA registers for direct addressing of the cache (we show only word accesses, but half-word and byte accesses are handled analogously). Software places values in the DA registers as an optional side-effect of performing a regular load or store. A tag-unchecked load or store specifies a full effective virtual address in addition to a DA register number. If the DA register is valid, its contents are used to avoid a tag search; if it is invalid, hardware falls back to a full tag search using the entire virtual address. The valid bit can be checked early in the processor pipeline, so this check does not introduce any additional latency into the cache access path.

```
Old Code            New Code
sub $sp, 64         sub  $sp,  64
sw  $ra, 60($sp)    swlda $ra, 60($sp), $da0
sw  $fp, 56($sp)    swda  $fp, 56($sp), $da0
sw  $s0, 52($sp)    swda  $s0, 52($sp), $da0
```

Figure 1.7. Code common at C function entry, and the same code transformed to use direct address registers. The `swlda` instruction writes `$da0` with the cache location of the data at virtual address `$sp + 60`. The `swda` instructions operate on that cache line without powering up the cache tags.

4.2 Example Use

As a concrete example, consider the code in Figure 1.7, common at C function entry, and a transformation of that code which uses direct addressing.

The direct addressed operations use `da0` which is set up by the `swlda` instruction. This allows the compiler to use the `swda` instructions to eliminate cache tag checks on up to 7 stores (the remainder of the cache line started by the store) without adding additional instructions. The compiler keeps the stack 32-byte aligned to support this transformation.

4.3 DA Register Implementation

At minimum, a DA register needs only to record the matching way within the cache set. In this case, the effective address is used to obtain the subbank number, the set index, and the offset within the cache line. In some implementations, however, it will be advantageous to also record subbank and set index information in the DA registers and to physically distribute the DA registers among the cache subbanks. This avoids recalculating and retransmitting these portions of the virtual address for tag-unchecked accesses. Further implementation details are given in [12].

4.3.1 Keeping DA Registers Coherent. The DA registers must be kept coherent with the state of the cache. If a line pointed to by a DA register is evicted, the DA register contents are no longer valid and cannot be used in a tag-unchecked access; i.e., the inclusion property between the DA registers and the primary cache must be preserved. Lines may be evicted either as a result of cache line replacement or by external invalidate requests to maintain cache coherence with DMA I/O traffic or other processors.

The primary cache can be used as a filter for snooping invalidations to the DA register, and the DA registers are only searched when a cache eviction occurs. Searching the DA tags consumes some additional energy on each evict, but it is only a small addition to the total cost of the cache line replacement which might involve fetching a line from DRAM.

4.4 Compiler Analysis

The general compiler algorithm that is used to eliminate tag checks is straightforward. Find two references, one of which dominates the other (so all paths that cause the subordinate access to be executed cause the dominant reference to be executed first). If we can prove that the two references always point to the same cache line, the second reference can skip the tag check. It is made tag-unchecked by having the dominant reference write a direct address register that the subordinate register reads.

Code between the two references, including assignments, control flow or even function calls, does not affect correctness because hardware will invalidate the DA register if the line gets evicted between the definition and the use of that DA register (as discussed in section 4.3.1). However, the compiler must analyze pointer assignments to prove that two references always point to the same location.

The compiler controls the stack pointer and so can ensure it is always aligned to a cache boundary. This allows easy transformation of function entry/exit code (like in Figure 1.7), spill code, and parameter passing code. Small automatic variables are never allocated across stack cache line boundaries, so references to local variables and spill code profit from use of the DA registers. Heap and static data require special compiler passes to restructure and analyze loops. Further details of compiler techniques are given in [12].

After all potential pairs are identified, each dominant reference that has a successful subordinate reference is a DA register candidate. The next task is to map these candidates onto the limited set of DA registers. This is a standard register allocation problem—DA register candidates that are live at the the same program point interfere and need to be allocated to different registers. The DA register allocation problem is simpler than processor register allocation because a DA register can not be spilled. Instead of spilling, a DA register is simply not allocated to a problematic DA variable.

4.5 Evaluation

The tag-unchecked compiler analysis was implemented for the SUIF compiler [5], which was configured to output instrumented C code. The instrumented code has loops unrolled and is augmented with statistics gathering capability. Figure 1.8 shows how many tag checks were eliminated and whether the elimination was for a load or store. It is important to break these cases out since the tag check is less of the total energy of a store since the value update takes energy.

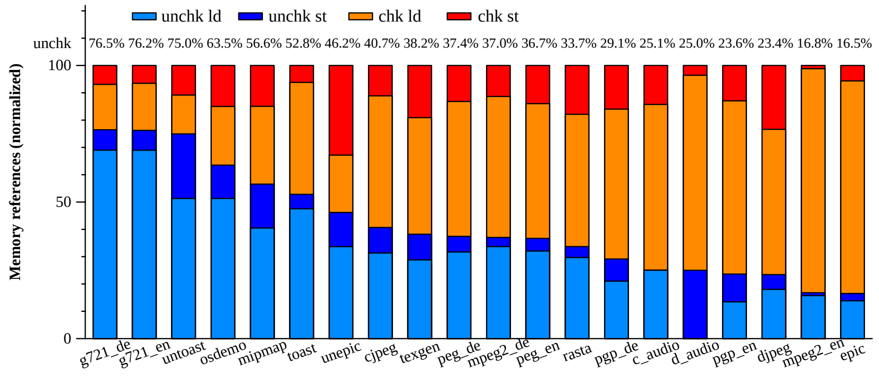


Figure 1.8. Tag check elimination for Mediabench programs compiled by SUIF. Eight direct address registers are used. The lowest part of the bar is tag unchecked loads, then unchecked stores. Over that are tag checked loads and stores. The number on top of each bar (unchk) is the percentage of tag checks eliminated.

5. Discussion and Future Work

One concern with exposing detailed internal machine state is that it can seriously constrain the design of future ISA-compatible implementations. In some embedded application areas, it is already accepted practice to change underlying ISA design frequently where there is a significant cost, performance, or power advantage. In other cases, dynamic translation of a virtual machine ISA alleviates the need for native machine compatibility (for example, Transmeta ships two incompatible VLIW ISAs to which x86 code is dynamically translated). Further development should make possible new energy-exposed ISAs that are portable across a range of implementations, and several of these features could be added as extensions to existing ISAs.

5.1 Instruction Chains

Explicitly targeting bypass latches would not work in a processor that reorders instructions, as the latches are typically overwritten every cycle. For example, in the following sequence:

```
add RS, r1, r2
sub r3, RS, r4
slt r5, r6, r7
```

if the `slt` instruction is issued after the `add` instruction but before the `sub` instruction, it will clobber the value in the RS bypass latch as it proceeds down the pipeline. One way to make use of exposed temporary state in an out-of-order processor is through the use of *instruction chains*. An instruction chain is a string of instructions where each instruction uses the result of a previous

gcc	**gsm(encode)**	**jpeg(decode)**	**pegwit(decode)**
sltiu,beq (9.7)	addiu,subu,sll, addu (30.6)	sll,addu (16.4)	sll,addu (18.8)
lui,lw (9.2)	sll,sra (15.6)	addu,addiu,sra (9.0)	srl,andi (9.0)
addiu,bne (8.2)	addu,sltu,beq (7.2)	addu,addu (8.9)	lui,addu,lw (9.0)
addu,lw (7.3)	mflo,addiu,sll,sra, addu (6.7)	sltu,bne (4.6)	addu,lw (6.8)
sll,addu (4.3)	sll,sra,mult (6.5)	lw,or (4.6)	xor,xor (6.1)
addiu,beq (4.3)	slti,bne (4.2)	lw,addu,addu (4.5)	andi,sltiu,bne (6.0)
slt,bne (3.2)	mflo,addu,sw (3.4)	addu,sra,addu, addu (4.5)	andi,bne (5.5)
lui,addiu (2.9)	addu,addu,addu (2.4)	sll,addu,sll,subu (3.3)	andi,sll (5.4)
slti,bne (2.9)	sll,addu (1.8)	addu,addu,sra,andi, addu (3.2)	andi,addiu (4.9)
slti,beq (2.4)	addu,addu (1.6)	subu,addu,sra,andi, addu (3.2)	andi,sltiu (4.7)

Table 1.2. Most frequent instruction chains for selected benchmarks. The numbers in parentheses are the percentages of all chains in the dynamic program execution represented by the corresponding instruction sequences.

instruction. Data dependencies require that instructions in a chain must be processed in order, but separate chains can be processed out of order. Essentially, the chain is viewed as a single large instruction by the reordering mechanism.

Our simulations show that on average about 41% of all dynamically executed instructions were in chains of at least two instructions; this indicates that the concept of instruction chains warrants further investigation. For example, when designing a new instruction set, chains could be incorporated into the architecture. CISC architectures effectively encode a few types of instruction chains as addressing modes for operands. Table 1.2 lists the frequently occurring chains in programs, and shows the considerable variety in the instruction chains generated for each program. These results indicate that a small number of instruction patterns will not suffice when creating a new instruction set — more generality is needed. Exploiting instruction chains should also allow a more compact instruction set encoding by using implicit chain operands, saving further energy in instruction fetch.

6. Conclusion

Instructions perform many hidden microarchitectural operations as they execute. Compile-time analysis can statically determine that much of this work is unnecessary. By providing an energy-exposed instruction set, this information can be communicated to the hardware to save energy without impacting performance.

One potential difficulty with exposing more machine state to the compiler is that it can potentially increase energy consumption due to exception management overhead. Software restart markers reduce this overhead by enabling the introduction of temporary state that does not have to be saved and restored across exceptions. Exposed bypass latches are an example of allowing software to make use of temporary state to avoid microarchitectural operations at run time; in this case register file reads and writes are statically eliminated. An instruction interface with alternative mechanisms to perform the same task allows a compiler to deactivate unnecessary portions of the machine microarchitecture. Tag-unchecked loads and stores are an example which use compile-time analysis to access the cache with direct address registers instead of costly tag checks.

The three energy-exposed instruction set features presented here will yield greater savings with more sophisticated global compiler analyses, and further energy-exposed instruction set features are under development. These techniques are just a first step towards future energy-efficient instruction set architectures.

References

[1] Furber, S. B. *et al.* ARM3 - 32b RISC processor with 4kbyte on-chip cache. In G. Musgrave and U. Lauther, editors, *Proceedings IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration (VLSI'89)*, pages 35–44. Elsevier (North Holland), 1989. ISBN 0 444 88344 4.

[2] Intel Corp. *Intel Xscale core developers manual*, order no. 273473-001 edition, December 2000.

[3] G. Kane. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1989.

[4] R. Krashinsky. Microprocessor energy characterization and optimization through fast, accurate, and flexible simulation. Master's thesis, Massachusetts Institute of Technology, May 2001.

[5] Lam, M. S. *et al.* The SUIF compiler system, 1992–2001. `http://www-suif.stanford.edu`.

[6] Montanaro, J. *et al.* A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal Solid-State Circuits*, 31(11):1703–1714, November 1996.

[7] M. Muller. Power efficiency & low cost: The ARM6 family. In *Hot Chips IV*, August 1992.

[8] V. Oklobdzija. Architectural tradeoffs for low power. In *Power Driven Microarchitecture Workshop at ISCA98*, Barcelona, Spain, June 1998.

[9] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proc. 12th ISCA*, 1985.

[10] J. Tseng and K. Asanović. Energy-efficient register access. In *Proceedings of the 13th Symposium on Integrated Circuits and System Design*, pages 377–382, Manaus, Amazonas, Brazil, September 2000.

[11] W. Walker and H. G. Cragon. Interrupt processing in concurrent processors. *IEEE Computer*, 28(6):36–46, June 1995.

[12] Witchel, E. *et al.* Direct addressed caches for reduced power consumption. In *MICRO 34*, Dec 2001.

[13] M. Zhang and K. Asanović. Highly-associative caches for low-power processors. In *Kool Chips Workshop, Micro-33*, December 2000.