The Span Cache: Software Controlled Tag Checks and Cache Line Size

Emmett Witchel and Krste Asanović
MIT Laboratory for Computer Science, Cambridge, MA 02139
{witchel|krste}@lcs.mit.edu

Abstract

The *span cache* is a hardware-software design for a new kind of energy-efficient microprocessor data cache which has two key features. The first is *direct addressing* which allows software to access 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. The second key feature is software controlled line size. This lets the compiler specify how much data to fetch on a miss, allowing greater cache utilization and reducing memory bandwidth requirements. Two possible hardware implementations of software controlled line size are sketched and discussed.

1 Introduction

Caching is one of the most effective techniques for increasing performance and decreasing energy consumption. But any given hardware implementation of a cache has to balance out many different, often incompatible, usage patterns. In this workshop paper, we present ongoing work to develop a new hardware-software interface for data caches that allows software greater control over cache operation to reduce energy consumption and increase performance.

One component of the span cache interface is *direct addressing* which allows software to access 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. When the compiler has the information, the tag check can be eliminated. But direct addressing gracefully degrades to conventional tag-checked accesses when the compiler cannot eliminate the tag check, or in the presence of interrupts or cache invalidations. Initial experiments with a Java compiler for SPECjvm98 code show that 13–34% of all data cache tag checks can be eliminated, saving 6-17% of cache access energy and 1–3.5% of total processor energy over a baseline low-power processor design. With better compiler analysis we expect to improve these results.

Software controlled line size lets the software specify the cache line size for each access. Because direct addressing eliminates many tag checks, it reduces the relative penalty of introducing a more sophisticated cache search scheme for cases where tag checks cannot be eliminated. Direct addressing helps enable software controlled line size. Software-controlled cache line size will allow the compiler to make better use of the cache. If the compiler knows the application only needs one word, only one word is fetched, and only that word and its tag reside in the cache. In a conventional cache, every word access brings in an entire line. Greater cache utilization means increased hit rates for a given cache size, or it can mean maintaining a given hit rate while reducing cache area.

Increasing hit rates reduces power consumption since misses need to go off chip which consumes a lot of power. Decreasing the cache area has many beneficial effects including cost reduction, and counteracting the effects of technology scaling which limits the size of cache that can be accessed in a fixed number of processor clock cycles [1]. It also reduces cache leakage current for a given hit rate, and reduces total memory access energy.

The span cache design takes into account the fragility of compiler analysis, and ensures that the hardware can act as a backup. For instance, one piece of code may reference an in-cache data item using direct addressing, but another piece of code can still use a conventional virtual address to find the same data item in the cache. Another nice feature is that a hardware implementation can chose to ignore direct addressing features altogether.

We focus on 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, e.g., [16, 17, 15].

We first review the design of a low-power cache. Then we explain caches tagged with content-addressable memory. In section 3 we discuss direct addressing, the process which allows us to avoid tag checks, and we present a thorough evaluation of the design in the next section. In

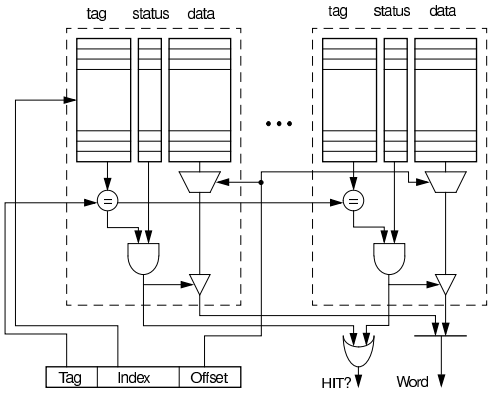

Figure 1: The organization of a set-associative RAM-tag cache. On every access, data is read out of every way, though the data from at most one way is used. This extra work needlessly consumes power.

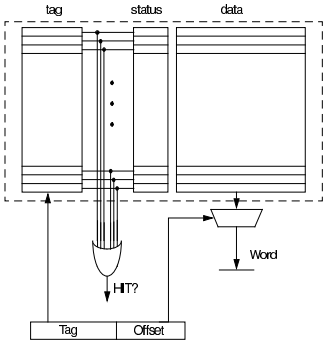

Figure 2: The organization of a highly-associative CAM-tag cache. The tag bits are broadcast to the CAM, and if there is a hit, the word is read out. The tag check is a high percentage of the energy cost, but the overall energy cost is roughly equivalent to a two-way set associative cache, and the miss rate is lower.

section 5 we describe software control of cache line size, and sketch two possible implementations of software controlled line size, one using just (S)RAM, the other using a content-addressable memory.

2 Current cache design

An energy-efficient cache design needs to find the proper cache associativity. Standard caches hold data and tag information in the RAM of a cache line. The hardware finds the data based on the virtual address, reads the data and checks the tag against the value stored in the line. The tag for a virtually indexed cache (which is common in energy-efficient designs since they do not access the TLB on a primary cache hit) consists of the upper bits of the virtual address and an address space identifier, which is unique to a process (while none of our techniques rely on having a virtually indexed cache, we will assume one for simplicity of exposition). An n-way associative cache does n tag checks in parallel. It also does n data reads in parallel, throwing out the value of all but one of them. This design is shown in Figure 1. While associativity is good for performance and for lowering miss rates, the redundant work it requires has a high energy cost.

Direct-mapped caches have lower hit energy because they only read one tag and one data word. But they have much larger miss rates due to conflicts. Since the energy miss penalty is large, they have larger total memory access energy [21]. Way-predicting caches can provide associativity at lower hit energy by only checking one way in an n-way set associative cache, but incur energy and delay penalties to access the way-prediction table on way hits and additional energy and performance penalties if predictions are incorrect [10].

Caches are also often split into subbanks, which are smaller, replicated caches which handle certain address ranges. Bank numbers are direct mapped using the appropriate virtual address bits. For a 16 KB cache with 1 KB subbanks, bits eleven through nine are the subbank number.

An alternative to RAM-tag caches, chosen by the StrongARM designers [6] and other energy-conscious designs [2] is to store the tags in content-addressable memory (CAM). Here the tag is broadcast to the cache lines, and only the line that matches has its data read out. The energy consumption of a 32-way CAM-tag search is approximately the same as a 2-way set associative RAM-tag search [21, 2]. CAM-tag caches are often subbanked, and one bank of the design is shown in Figure 2. Although CAM-tag caches reduce miss rates and hence total access energy, they expend relatively greater energy in tag checks.

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. For our HSpice simulations of our CAM tagged cache, the tag check consumed 43% of cache energy for stores, and 54% of cache energy for loads, for a 16 KB cache with 1 KB subbanks. 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.

3 Direct addressing

In order to eliminate the tag check and data search of a CAM-tag cache, we want to change the processor interface from issuing virtual addresses to using something that tells the hardware exactly what cache line has the needed data. We believe this is best expressed as the contents of an on-cache register.

In our proposed design, we augment the cache state with eight registers, called direct address (DA) registers. We arrived at the number eight experimentally. These registers contain enough information to specify the bank and way in the line used in that access. The exact width and data layout of the register is implementation dependent and never made visible to software. In a tag-unchecked access, the CPU specifies a DA register number, and the hardware uses the register's contents with the offset from the virtual address. Since the hardware is unconstrained in the layout of the DA registers, they can be implemented efficiently using the natural layout of the intermediate bit vectors generated for regular cache accesses. The DA registers are physically distributed around the cache layout with each individual bit field held close to the portion of the cache that requires it. In essence, the DA registers memoize the results of a cache lookup using latches place alongside the various cache address and control signals. We expect minimal additional delay to mux in a DA signal versus a regular cache access signal.

Table 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). One flavor of memory operation does its memory operation and writes a direct address register, the other does its memory operation using the direct address register.

There should be no performance impact from using direct addressing. Direct addressing is an optimization and even allows a null implementation where all direct address register information is simply ignored.

3.1 An example of direct cache addressing

As a concrete example, consider the code in Figure 3, 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. Our compiler keeps the stack 32-byte aligned, which allows this transformation.

```
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 3: Code common at C function entry, and the same code transformed to use direct address registers.

3.1.1 Alignment assumptions

Alignment information is needed for the compiler to use DA registers effectively. The compiler controls the stack pointer and so can ensure it is always aligned to a cache boundary. Small automatic variables are never allocated across line boundaries, allowing references to local variables and spill code to profit from use of the DA registers.

For heap-allocated data, there are two options depending on the source language. For languages that feature automatic memory reclamation, such as Java, we can modify the system allocator to follow some alignment policy. All memory comes from the system allocator, so we have global guarantees on the alignment of data.

For languages like C, static compile-time analysis is more difficult and in our scheme we rely on profile information to get predictions of expected alignment. Where we expect to see data aligned, the compiler generates two copies of the code and chooses between them with a run-time alignment test. If the test succeeds, we execute an optimized version that uses the alignment information to eliminate tag checks. If the test fails, we execute the vanilla compiled version.

This type of optimization is most profitable in loops where it can be hoisted out of the loop body and folded into the checks normally done for loop unrolling. Unrolling is done for both performance gain, and for tag-access elimination. One disadvantage of using loop unrolling to obtain alignment information is that too much unrolling can increase I-cache pressure [14].

3.2 Coherence

The DA registers must be kept coherent with the state of the cache. The coherence actions for line replacement and for external intervention (e.g., for DMA or cache-coherence in a bus-based system) are the same. On any eviction, we perform an associative search of the DA registers to see if any are pointing to the victim line. If so, we invalidate the DA register, preserving the inclusion property between the L1 cache and the DA registers. This allows us to use the L1 cache as a filter for snooping invalidations to

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|w)wlda`, accessing memory and setting the `da` register.
`daflush mask`	Flush direct address registers specified by the bitmask (which is little endian). This clears the valid bit on the specified direct address register. This instruction is used at the end of function calls and by the operating system when the DA register lifetime has ended.

Table 1: A table of instructions for manipulating direct address registers

the DA registers. We only search the DA registers if the snoop causes an eviction.

Searching the DA registers consumes some additional energy on each evict, but it is only a small addition to the total cost of the replacement which might involve fetching a line from DRAM. We can reduce the cost of searching the DA registers by using a conservative scheme that considers fewer bits of the address, at the cost of some additional spurious invalidations.

3.3 Operating system maintenance of inter-process protection

A similar issue is how the operating system maintains DA registers state when a process is descheduled. We have two choices for maintenance of the DA registers. The first is for the OS to save and restore their value. We briefly discuss the reasons for not doing this.

To save and restore the DA registers, the OS could not simply save the bit patterns stored in the registers. The bit patterns in the registers are direct cache addresses and they point to lines in the cache. When the OS reschedules the process the contents of the cache are unknown, and unlikely to be the same as when the process was last running. The previous DA register state is therefore not useful to the process and indeed might point to lines the process has no right to access.

In order for the OS to save and restore the DA registers, it would also have to recreate the part of the state of the cache expected by the process about to run. In order to do that it needs the virtual address the process used to set up the DA registers. This is not stored anywhere. We don't want it in the DA register itself since that makes the hardware more complicated. An awkward solution would have the compiler output a table telling the operating system for every program point what DA registers are in use with what virtual address. Maintenance of this table is too complicated to justify, and the instruction cost to access it would be hundreds of times the cost of an invalidation based scheme.

The second option is for the OS to explicitly invalidate all of the DA registers in between process contexts. This is the option we choose. The OS invalidates any DA register before it uses them (of course the OS is free to use the registers for its own code), and invalidates all of them before the next user context is run. This enforces inter-process protection because valid entries are never communicated between protection domains.

With this invalidation-based scheme, a process that was descheduled while it was using the DA registers will be rescheduled by the OS with all of its DA registers invalidated. When it tries to use a DA register, it will cause a regular tag-check to be performed using the full virtual address, and set the DA register to a valid value. Being descheduled only means the process will incur a full tag check cost once for each DA register it was using; direct addressing gracefully degrades to standard tag checking.

3.4 Separate compilation units

Most compilers analyze one function at a time, and DA registers are like hidden parameters. It would require inter-procedural analysis for a function to know that e.g., a DA register pointed to some globally visible data. We do not

implement inter-procedural analysis, so our compiler just invalidates all DA registers used in a given function at that function's return. DA registers that are live across function calls are in danger of being invalidated (making their next use a tag-checked operation), if they are used by the called function.

We have not observed this to be a big problem because function calls in tight loops are rare since they are also bad for performance. So we currently naively allocate DA registers from zero to seven. If interference from function calls becomes an issue, it is possible that some simple allocation policy, like non-leaf procedures allocating from zero to seven, and leaf procedures allocating from seven to zero, will avoid most problems from function call invalidations.

4 Status and results for direct addressing

We have implemented the compiler support for tag check elimination in FLEX, a Java bytecode-to-native compiler developed at MIT [7]. FLEX takes Java byte codes and produces a MIPS-like assembly language that has support for direct addressing. We modified the GNU gas assembler to accept this assembly language, and we generate MIPS-like binaries which run in our simulation system. The numbers in this section come from a standard MIPS ISA simulator with cache model, modified to perform sanity checks on our direct addressing instructions (so that e.g., the virtual address was always in the line referenced by the DA register). We are currently implementing improved versions of these algorithms in the SUIF C compiler from Stanford [8].

We analyzed user code to find accesses to objects that dominate other accesses, and transformed the subordinate accesses to use tag-unchecked accesses. Access A dominates access B if and only if every execution path that includes B includes A as a predecessor. We modified the heap allocator to not split objects over cache lines, so the compiler knew that if an object was smaller than a line, all references to it were to the same cache line.

Our implementation for user data was limited. We only included a single direct address register, and we only allowed that register to be live within a single basic block. We plan a more aggressive implementation.

One advanced feature of the Java implementation was our transformation of spill code. We modified the standard MIPS calling convention for C to pack callee-saved registers with the return address and frame pointer on the stack. By packing them onto a line, we can usually use a single direct address register for the save and restore.

Some data for the reduction in tag checks is shown in Table 2. All benchmarks were run using the medium sized input (size 10) from the Java context structure. This size allows the benchmarks to enter their steady state, but limits simulation time.

These numbers are surprisingly high given that we only used one direct address register which is only live in a single basic block. Spill code accounts for a large portion of the savings. Our preliminary C implementation only transforms user references, ignoring spill references, and we are seeing around 50% tag check reduction using eight DA registers on Mediabench applications.

To get the reduction in cache power, we use our model of a low power cache and processor. Our figures come from a SyCHOSys simulation [13], which uses a transition-sensitive detailed structural model. The tag search is 57 pJ, out of a total 106 pJ for loads (54%), and 133 pJ for stores (43%). This is a highly optimized CAM-tag cache with 1 KB subbanks, segmented word lines, and low-swing bitlines. Saving additional energy over this optimized baseline is difficult.

The reduction in total processor plus cache energy is very sensitive to the details of the processor design, and somewhat sensitive to the benchmark. We show an estimate based on the average energy consumption spent checking the data cache tags. This estimate came from detailed, transition sensitive simulations of our processor model executing Mediabench and SPECint95 [12]. The variation from benchmark to benchmark was reasonably small, so an aggregate figure of energy consumed by data cache tag checks is reasonable. [12] measures this average energy consumed at 10%. Our tag check elimination saves part of that 10%. The control logic for direct addressing, and the decode energy for the extra instructions will consume extra energy, and we do not model that effect.

Our reduction in tag checks yields a 6–17% reduction in data cache energy, and an overall savings of 1–3.5% for the entire system. The energy saving is highly dependent on the processor and cache implementation. For example, in our initial system design the cache had 2 KB subbanks, resulting in a larger overall saving of 3–8.5% (and data cache energy savings from 13.7–34.6%). But the smaller 1 KB subbank we chose for our baseline reduces total access energy, at the cost of an increase in cache area.

These initial results are encouraging and we believe the more aggressive compiler analysis schemes we are developing can reduce energy further. But perhaps more important than the raw energy savings is the idea that eliminating tag checks can open the door to more complicated tag search schemes, since tag checks need not be performed on every reference.

Benchmark	inst count	Tag checks eliminated ld	st	ld + st	cache energy reduction	Processor + cache energy reduction
Compress	47582324	28.1% (2410368)	46.5% (2105908)	34.6%	17.0%	3.5%
db	338223648	14.8% (9277168)	26.9% (9224657)	19.1%	9.4%	1.9%
Jess	105221820	13.8% (2310078)	24.7% (2553688)	18.0%	8.8%	1.8%
Jack	671905543	10.2% (12755070)	20.0% (14073950)	13.7%	6.7%	1.4%

Table 2: SPECjvm98 programs with the percentage (and count) of tag checks eliminated using a single direct address register, live in a single basic block. The higher store numbers are due to the aggressive transformation of spill code. We present energy savings relative to just the data cache, and the entire instruction and data cache and processor system.

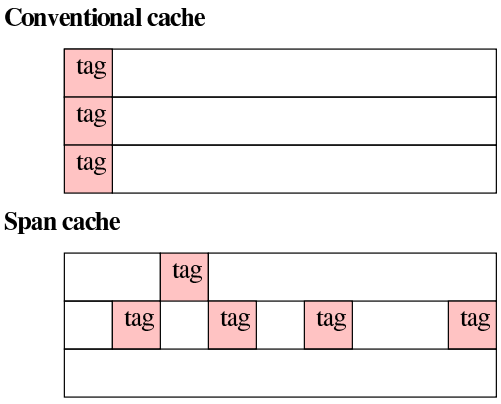

Figure 4: On top is a conventional cache structure with fixed place tags and fixed offset and length data. Below it is a span cache structure with variably placed, word-length tags and variable length data. In both figures, the tag area is shaded, the data area is clear. The maximum length of data in the span cache is constrained by a particular implementation.

5 Controlling line size in a span cache

The goal of the span cache is to make better use of the cache—either by increasing hit rates for a given cache size or by maintaining a given hit rate while shrinking cache area. The fixed size and alignment of traditional cache lines conflicts with the memory access patterns of applications, preventing software from fully utilizing the cache area. By giving software control over the line size on a per-access basis, the software can more effectively manage the hardware.

Traditional caches move data in lines of fixed offset and length. The span cache generalizes this notion of lines into spans. A span is some number of words at some offset. Software can control the span parameters when it has knowledge enough to do so. Figure 4 shows a comparison between a traditional cache structure, and the span cache. The challenge for the compiler is finding enough information about references to make the software control worthwhile. The challenge for the hardware is a circuit design that is fast and allows this flexibility.

The span cache uses more hardware support to give the compiler the ability to determine the size and placement of data in the cache. Hardware tag schemes are simple for two reasons—the hit case should happen in a single cycle, and the tags are checked on every access. We want to maintain single cycle hits for performance, but if tags no longer have to be checked on every access, that alleviates some of the need to keep tag schemes very simple. Fixing the offset and length of cache lines makes them fast to access and easy to implement, but it wastes a lot of cache area, as the measurements in Table 3 show.

Table 3 shows a story familiar to computer architects. Some programs, like `ijpeg`, stream through data with unit stride and hence completely use cache lines. These applications benefit from long cache lines since long lines help the application exploit spatial locality. Applications with more complicated data structures and more chaotic access patterns like `gcc` and `m88ksim` show a more bimodal behavior with many lines having only one word accessed, and many lines having all eight words accessed. This data was collected on a CAM-tag cache with 1 KB subbanks, 32-way set associativity and a FIFO replacement policy. The StrongARM primary cache also has a FIFO replacement policy.

We propose allowing software to specify both the location of each access and also the size and shape of the region of memory being accessed (within some hardware limitations of course). We will present the software interface, then examine two possible hardware implementations.

5.1 Software interface to a span cache

No changes are necessary for software to use a span cache. Regular loads and stores default to cache lines of traditional size and alignment (which is 32 bytes, 32-byte aligned in our design). Direct address registers can be used with this default size and alignment information as well. Span caches distinguish themselves by allowing software to specify the offset and length of the memory region

Benchmark	1–4 words	5–8 words	1 word	8 words
pegwit_enc	96.5%	3.5%	82.2%	2.6%
gcc	51.7%	48.3%	16.9%	33.3%
m88ksim	69.8%	30.2%	61.1%	23.0%
ijpeg	16.8%	83.2%	6.9%	74.0%

Table 3: SPEC INT 95 and Mediabench programs simulated with a 16 KB, 32 byte line primary data cache. For every line evicted, the table shows the percentage of lines that had less than half a line accessed (1–4 words), and the percentage that had more than half a line accessed (5–8 words). We also present the percentage of lines that had only 1 word or all words accessed as these are important special cases. Data streaming applications like ijpeg tend to fully utilize cache lines while complicated integer codes look more bimodal with many lines fully used and many lines with only one word used.

fetched on a miss for a particular access. The maximum length of a span is 64 bytes in our design.

Each reference for a span cache can specify a virtual address, direct address register, and shape. The shape is the shape of the span to be fetched if this access misses. It consists of an offset from the virtual address specified in the instruction, and a length.

The standard (non-DA) load and store instructions access the 32 byte, 32-byte-aligned cache line on which the reference occurred, providing the same behavior as a conventional cache scheme.

5.1.1 Cache line shape encoding

Specifying a span requires one byte. We have a maximum line length of 16 words (64 bytes). The minimum line length is a word, so we require 4 bits for a length field(1–16). For full generality, we require a 4-bit negative offset (-15–0) which specifies where the line is supposed to start relative to the current word. So an offset of -2 means this word is the third in the line. This encoding allows an access to be anywhere in a span of up to 16 words.

We are currently investigating several alternative schemes for encoding this information in the instruction stream. The most promising approach is to use variable length instructions, as these have a number of additional advantages in a low power setting. Instruction fetch consumes a lot of processor power, and some instructions are more popular and easier to encode than others, so a variable length encoding, like the 16-bit, 32-bit, and 48-bit instructions in the IBM 360 architecture, can significantly reduce static code size and instruction cache bandwidth while easing the addition of new instruction forms.

Since the shape information is only needed on a miss it would also be possible to set it up in advance if we wished to restrict ourselves to a fixed-length instruction set. For instance, an additional instruction could set the shape information for references via a given DA register. We need more information about how these addressing options interact before we can know what would be the best encoding.

5.1.2 Using the span cache

Assuming that we simply tack the offset and length information onto the memory access instruction, Table 4 is a code example in C showing how this interface can be used.

The code example shows the common interplay of word oriented and array oriented data. The global struct control has a frequently used result field, and some infrequently used error fields. Accesses to the result field are performed specifying an offset of 0 words and a refill line size of 1 word. Accesses to the data array specify a refill of the fetch address and the next 15 words (16 words total). Since the access to data[i+1] occurs before the access to data[i], a miss on data[i+1] starts its miss refill at one word before its address to include data[i].

The loop is unrolled to maintain alignment information. There are still checks between unrolled instances of the loop so there does not have to be an additional fix-up pass for any remaining iterations. Notice that the last access to A[i+1] is accessing the next line, and so cannot use `$da1`, it must use a regular load.

In a conventional cache, this loop would execute 80 tag checks on the `data` array. Our version executes 10 tag checks on it, assuming it was cache aligned on entry to the function, if not, 11 checks are performed.

`ctl->result` gets register allocated, though if there were other pointer writes in the loop it would be likely that alias analysis would fail and it would not be register allocated. If the compiler could not register allocate `ctl->result`, we would still be able to eliminate the write tag check in the read-modify-write that happens on each loop iteration, cutting the tag check count for that field in half.

In a conventional cache, this loop would fetch five 32-byte lines from the data array, and one 32-byte line from the control structure. Using a span cache, this code would fetch three 64-byte spans from the data array and a one word span from the control structure.

We assume that the backing store for the miss (DRAM in our design) will only transfer the words needed for a particular access. Thus span caches not only increase cache utilization, they also reduce memory bandwidth requirements. A conventional cache could use per-word dirty bits to reduce bandwidth on write-backs, but the span cache can also reduce fetch bandwidth.

C code derived from gsm in Mediabench

```
struct control {
    int error_code[2];
    int result;
};
void sample(struct control* ctl, int* data) {
    ctl->result = 0;
    for(int i = 0; i < 39; ++i) {
        ctl->result +=  data[i] * -134 + data[i+1] * -374;
    }
```

Machine code equivalent using a span cache

```
_sample:
  li     $t3, 0                    # ctl->result = 0
  add    $t8, $a1, 4 * 39          # t8 == loop bound
_for_loop:
  lwlda  $t0, 4($a1), $da1,-1, 15  #
  mult   $t2, $t0, -374            # t2 = data[i+1] * -374
  lwda   $t0, 0($a1), $da1, 0, 15  #
  mult   $t1, $t0, -134            # t1 = data[i] * -134
  add    $t3, $t3, $t1             #
  add    $t3, $t3, $t2             # ctl->result += t1 + t2
  add    $a1, 4                    # ++i
  bge    $a1, $t8, _exit           # loop test
  lwda   $t0, 8($a1), $da1,-1, 15  #
  mult   $t2, $t0, -374            # t2 = data[i+1] * -374
  lwda   $t0, 4($a1), $da1, 0, 15  #
  mult   $t1, $t0, -134            # t1 = data[i] * -134
...another 5 copies of the body are unrolled..
  lw     $t0, 32($a1),      -1, 15 #
  mult   $t2, $t0, -374            # t2 = data[i+1] * -374
  lwda   $t0, 28($a1), $da1, 0, 15 #
  mult   $t1, $t0, -134            # t1 = data[i] * -134
  add    $t3, $t3, $t1             #
  add    $t3, $t3, $t2             # ctl->result += t1 + t2
  add    $a1, 4                    # ++i
  blt    $a1, $t8, _for_loop       # loop test
_exit:
  sw     $t3, 8($a0),       0, 1   # update ctl->result
```

Table 4: C code and its assembly, unrolled and optimized by hand to use a span cache. The `lwlda` after the `_for_loop:` label loads direct address register `$da1`, which is used by the `lwda` after the `mult` instruction. The `lwda` access is tag-unchecked. The third field of the access is the offset, and the last is the length. So the first `lwlda` instruction specifies a line that starts one word before `4($a1)` (i.e., at `0($a1)`) and is 16 words long.

In the next two sections we present prototype designs for a span cache. The first design is based on a RAM block, which reduces implementation area but has significant overhead on tag-checked access. The second design is based on a CAM block, which adds additional area overhead, but has single cycle tag-checked loads. Both designs enable direct-addressed accesses with access energy comparable to a plain scratch-pad RAM.

We have not yet done circuit design for either direct addressing or flexible line size hardware, but we have done

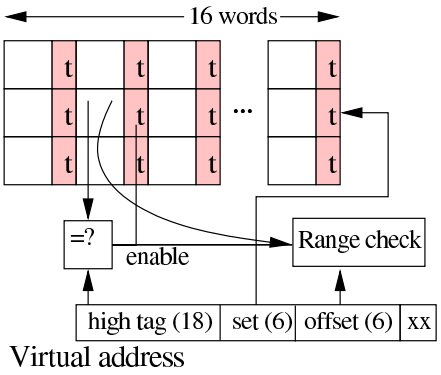

Figure 5: A span cache implementation in RAM. Each word has a bit indicating if it is a tag. All tags in a set are searched associatively in parallel, though the search circuitry is only shown for one word. The high bits of the address need to match the high part of the tag, the middle bits pick the set, and the low bits need to be range checked to see if this offset is contained in this span. Since data is stored in word chunks, the last two bits are not used. Finally, the cache has to mux out the correct word. The range check and mux make this cache more challenging to build than a traditional cache.

initial feasibility studies to ensure that both features are compatible with our current low-power cache layout.

5.2 A RAM-based span cache

Our first design for the span cache, shown in Figure 5, holds both tags and data in the same RAM. The RAM is broken into sets of 16 words each of which has a bit indicating if it is tag (the t-bits in the figure). The tags divide the set into spans of potentially different length. Every word following a tag (occupying a cell at a higher address) is the data associated with the tag. The data stops at the next tag or at the end of a line. All possibilities from one line of 60-bytes to 8 lines of 4-byte words are supported.

Figure 5 also shows how an address is looked up in the span cache (the circuitry for only 1 word is shown, but there is parallel circuitry for all 16 words). The middle bits of the address pick a set, just as in a direct mapped cache. Once the set is chosen two operations happen in parallel. The upper 18 bits of the address are matched against the upper 18 bits of the tag. In parallel, the low offset bits are range checked with the base of the tag and the length which is determined by finding the next set t-bit or the end of the set. These checks are only enabled if the t-bit is set for a given cell, and checks for all members of the set happen in parallel.

One disadvantage of the RAM implementation is that if the tag check succeeds, we then have to mux out the correct word for the CPU. This requires finding the offset of the given virtual address from the tag value, and hence an additional cycle of latency for cache hits. Like a conventional cache, the RAM implementation needs two cycles for a write (one cycle for tag check, followed by the data write).

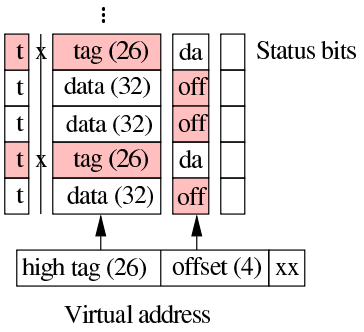

Figure 6: Part of one subbank of a span cache implementation using CAMs. Each word has a bit indicating if it is a tag. Bits 5..2 of the address are matched against the offset field (bits 1..0 are ignored since data is stored word aligned), while bits 31..6 are matched against the tag field. The tag field is matched if the t bit is set, otherwise the offset is matched. This is indicted by the shading in the figure.

We keep a rotating pointer within the sets, and replace spans in FIFO order. We therefore need to be careful about maximum span conflicts. Since associativity is only provided when we have small spans, using the maximum span size effectively makes the cache direct mapped.

5.3 A CAM-based span cache

The RAM-approach to building a span cache suffers from large overheads on tag-checked accesses. We are experimenting with a CAM-based design that uses a fully-associative search to locate a word with a cache line. Our design is shown in figure 6.

There are many similarities between the RAM and CAM implementations. Any word can be a tag or a data item. There is a high part of the tag, and a low part, but the partitioning is different, and both parts get searched in parallel, allowing single cycle reads. If the t bit is set, then the cell matches 26 bits of the virtual address against the tag, otherwise it matches 4 bits against the offset.

If a word is a tag, it broadcasts its hit signal down to subsequent data words. The hit signal broadcast chain is stopped by the next set tag bit. The data word ands the tag hit signal with its local offset match and on a hit the data is read out. Since the match logic is local to the word, we have single cycle reads and writes.

While we need comparators for the tag and offset checks, every tag must have at least one word of data, so each pair of words can share a single comparator. We require at most $N/2$ comparators for N words. We are still working on layout designs, but estimate around a 20% overall cache area overhead versus a conventional cache for this design.

From a software perspective, spans can be at any offset, and do not have to be naturally aligned (e.g., a 2-word span does not have to be 2-word aligned). This allows the compiler to use a span when it can not establish the alignment of a data item. But the hardware needs to keep tags for naturally aligned data, and so might split a single software span into multiple hardware spans. All of the data is still accessible.

5.3.1 Direct addressing

We leverage the CAM structure for direct addressing to remove the need for CAM banks to have a separate address decoder. The DA register points to the subbank where it is in use. We store the DA register number in the tag offset field. So a direct-addressed access does a 3-bit associative search on the DA register number, and uses the 5-bit associative search on the offset. A direct addresed access does not perform 26-bit associative search on the upper tag. When a DA register is invalidated, its number is broadcast to the subbank where it is in use so it can be invalidated.

Since direct addressing uses a small associative search, data words accessed from a direct address register do not need to be physically adjacent. This allows the hardware to break software spans into naturally aligned pieces without breaking software's view of non-aligned spans.

5.3.2 Incompatible, overlapping spans

One problem for the span cache is incompatible, overlapping spans. If we have a one word span at address 0x104, and we bring in a two word span at address 0x108, what happens? The data at 0x104 is in danger of being in two different places in the cache, and having different values. Referring to data by different shapes would only tend to happen when compiler analysis failed, so we do not expect it to be the common case. But it must be dealt with.

We have a simple, correct solution, and we present an optimization for the default case. On a miss, in parallel with sending the miss address to the DRAM (or second level cache), we search the primary cache for every word in the new span. Any clean matches are invalidated, and any dirty matches are read out and put in the write buffer. The entries in the write buffer, as always, are merged with the incoming data. In this way dirty data from the old span is preserved in the new span.

While the latency of a miss should give us enough cycles for a word by word search, the mechanism is unsatisfying since we expect conflicts to be so rare. An optimization for naturally aligned spans is that by using don't care bits, a single probe will determine if there is any data contained in the new span already in the cache. Since the default load and store shape is a 32-byte aligned block, we can tell with a single probe if there is no conflict. Only if the data is already resident do we need to go word by word.

6 Related work

There has been some related work on techniques to remove data cache tag checks. The ARM6 processor avoids tag checks for sequential accesses to the same cache line when using load multiple, and store multiple instructions. These instructions only need to check the tags for the first register read or written, but are typically only used for procedure call/return. Our model allows significantly greater flexibility.

Many architectures allow some limited form of software control of the cache. For instance the MIPS [11] has a cache control instruction which allows software to mark a line as invalid. Often the use of these facilities is not encouraged, e.g., the MIPS cache control instruction is privileged.

Software controlled caches are not a new idea though what parameters software has control over varies considerably. One early, extreme form of software control was in [3] where the cache refill engine is implemented in software, but here the large software overhead was hidden by using very long cache lines which are a poor match to many applications.

One strain of software-controlled caches allow software to partition the cache into different regions, often using already existing hardware for set-associativity. The reconfigurable cache project [18], and the column-cache [5] both exploit this technique. A more limited form of this partition is the StrongARM SA-1100's mini-cache [20] designed for data with only spatial locality. The span cache is orthogonal to these techniques, but we are investigating how DA registers can be used to control replacement policy.

Virtual lines [19] fetches data in large blocks from memory to hide access latency, but only moves small pieces into the primary cache as needed. This technique relies on excess bandwidth to fetch long lines from main memory, and would cause excess energy dissipation for accesses with little spatial locality.

One example of a specialized cache design is the vector data cache of the Cray SV1 system [9], which employs single word lines. Vector applications using strided

or scatter/gather accesses have limited spatial locality, and the single word cache design helps hold more data to capture any temporal locality. The span cache in contrast can support single word lines but also allows increased capacity for data regions that have significant spatial locality by reducing the amount of tag storage.

Several systems support boot time configuration of cache line size over a limited set of values (e.g., 32 or 64 byte lines). The line size is usually fixed at system design time and cannot be varied by software.

6.1 Application level alternatives to the span cache

There are a variety of application level techniques to compress user data structures. One survey [4] suggests several alternatives of varying levels of programmer difficulty. One common weakness of these approaches is that if access patterns change during the course of computation, then software reorganization can not express both kinds of locality. The span cache can adapt to different access patterns for the same data.

One technique is to augment malloc with another argument taking a pointer likely to be accessed when the pointer being allocated is accessed. This can require significant program understanding. In general, these solutions are difficult to automate for pointer-based codes since compiler analysis often fails, and the transformation requires whole program knowledge.

7 Summary

We have presented direct addressed caches as a hardware-software cache design that allows low power operation. Software gives hints to the hardware, based on its understanding of the program's access patterns, that allow the hardware to avoid powering up and searching the CAM tags. Even with simple compiler analysis, 14–35% of tag checks can be eliminated.

We have presented initial ideas for a span cache, that would increase the effective capacity of a cache by allowing software control over cache line size individually for each access. The additional delay and energy overhead of locating a data word in the span cache can be avoided by using direct addressing. We are working on more efficient hardware implementations of the span cache and on a complete evaluation.

8 Acknowledgements

This work was funded by DARPA PAC/C award F30602-00-2-0562.

References

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, June 2000.

[2] Tom Burd. Energy-efficient processor system design. *Ph.D. Thesis, Univeristy of California, Berkeley.*, 2001.

[3] D. R. Cheriton, G. A. Slavenberg, and P. D. Boyle. Software-controlled caches in the VMP multiprocessor. In *Proc. of the 13th Annual Symp. on Computer Architecture*, pages 367–374, New York NY (USA), 1986.

[4] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999. `citeseer.nj.nec.com/96477.html`.

[5] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Design Automation Conference*, pages 416–419, 2000.

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

[7] Martin Rinard *et al.* The flex compiler infrastructure. 1999–2001. `http://www.flex-compiler.lcs.mit.edu`.

[8] Monica S. Lam *et al.* The suif compiler system. 1992–2001. `http://www-suif.stanford.edu`.

[9] Greg Faanes. A CMOS vector processor with a custom streaming cache. In *Proceedings Hot Chips 10*, pages 103–110, August 1998.

[10] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. A high-performance and low-power cache architecture with speculative way-selection. *IEICE Transactions on Electronics*, 2000.

[11] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1992.

[12] Ronny Krashinsky. Microprocessor energy characterization and optimization through fast, accurate, and flexible simulation. *MIT Master's thesis*, May 2001.

[13] Ronny Krashinsky, Seongmoo Heo, Michael Zhang, and Krste Asanović. Sychosys: Compiled energy-performance cycle simulation. *Workshop on Complexity-Effective Design, 27th International Symposium on Computer Architecture*, June 2000.

[14] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. *ACM SIGPLAN Notices*, 33(11):46–57, 1998.

[15] Albert Ma, Michael Zhang, and Krste Asanović. Way memoization to reduce fetch energy in instruction caches. *ISCA Workshop on Complexity Effective Design*, July 2001.

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

[17] R. Panwar and D. Rennels. Reducing the frequency of tag compares for low power I-cache design. In *SLPE*, pages 57–62, October 1995.

[18] P. Ranganathan, S. Adve, and N. Jouppi. Reconfigurable caches and their application to media processing. In *Proc. of the 27th Annual International Symp. on Computer Architecture*, 2000.

[19] O. Temam and Y. Jegou. Using virtual lines to enhance locality exploitation. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, Manchester, England, 1994.

[20] Jim Turley. SA-1100 puts PDA on a chip. *Microprocesor Report*, 11(12):1,6–8, September 1997.

[21] Michael Zhang and Krste Asanović. Highly-associative caches for low-power processors. *Kool Chips Workshop, 33rd International Symposium on Microarchitecture*, 2000.