Revisiting Stack Caches for Energy Efficiency

Lena E. Olson [1] Yasuko Eckert [2] Srilatha Manne [2] Mark D. Hill [1]
[1] University of Wisconsin – Madison
{lena, markhill}@cs.wisc.edu
[2] AMD Research
{yasuko.eckert, srilatha.manne}@amd.com

Abstract

With the growing focus on energy efficiency, it is important to find ways to reduce energy without sacrificing performance. The L1 data cache is a significant contributor to processor energy consumption. We advocate treating data from the program's stack differently from non-stack data to reduce energy. We characterize stack accesses to determine how they differ from general memory accesses in terms of footprint, frequency, and ratio of loads to stores.

We then propose two ways to optimize for these characteristics. First, the implicit stack cache *limits stack data to residing in designated ways of the data cache, reducing the energy required per stack access. We show that it can reduce data cache dynamic energy by 37% with no reduction in performance.*

Second, the explicit stack cache *stores stack data in a separate L1 cache. In addition to reducing the energy per access, it also has additional benefits over the implicit policy in that it can be virtually tagged and have a different writeback policy. We show that this approach can lead to additional energy savings, with no performance impact. These optimizations are implemented purely in the hardware and thus require no changes to existing code.*

1. Introduction

With the growing focus on energy efficiency, it is important to find ways to reduce energy without sacrificing performance. One contributor to the energy consumption of the chip is the L1 cache. In Sodani's Micro 2011 Keynote [19], the fraction of core power that goes to caches was given as 12-45%, depending on whether the workload was floating point heavy or not.

One common strategy for saving L1 cache energy is to split the cache into separate data and instruction caches. The motivation for this is that there is no need to access both caches on every load or store; accessing a smaller structure is faster and takes less energy. In addition, this avoids the need to multiport both structures.

Similarly, it is possible to categorize data by whether or not it is from the program stack. The stack grows on function calls and shrinks on returns, and is used for storing local variables. Stack data has fundamentally different characteristics than non-stack data; we advocate taking advantage of them. We find that on average, 40% of memory accesses are to the stack, indicating that optimizing for them could significantly reduce energy. We characterize stack data and show that it differs from non-stack data in several important ways, including having high temporal locality, a small footprint, and a high proportion of writes compared to non-stack data. These characteristics are common to both x86 and ARM systems.

To take advantage of these properties, we must be able to distinguish stack and non-stack accesses. Because performance is highly sensitive to L1 data cache latency, and increasing the access time will significantly degrade performance, it is vital that we not add logic to the data cache access critical path.

We discuss several ways to predict whether a memory access is to the stack, including our preferred approach: testing whether any of an access's effective address components (e.g., base register and offset) are individually close to the stack pointer. Unlike previous proposals, the new approach does not add logic to timing-critical address-calculation path (as it is done in parallel with effective address calculation) and leaves unchanged the ISA, operating system, and applications. While mispredictions are allowed by our design (e.g., if two or more components are large), results show that the maximum number of mispredicted accesses due to comparing with effective address components or movement of the stack pointer is less than 1.2% in our workloads on x86_64 and ARM, and on average 0.2%. Because our optimizations do not rely on any characteristics of stack data for correctness, even in the case of misclassification the execution is correct.

Using this stack classification, we advocate treating stack and non-stack data differently in the L1 cache to improve energy efficiency. We discuss two ways to do this: an *implicit stack cache* and an *explicit stack cache*. The implicit stack cache requires only minimal changes, while the explicit stack cache is more complex but offers additional opportunities to save energy. Both approaches are invisible to the application and operating system, do not increase L1 access time, and are simple to implement.

First, the implicit stack cache modifies the data cache but does not add any additional structures. We change the replacement policy to ensure that stack blocks are only ever found in certain designated ways of the cache. Because the footprint of the stack is small, this generally does not

increase the miss rate for stack accesses. We are then able to optimize lookups of stack data so that they only check the ways where the stack blocks are expected to reside. We find that this approach can reduce L1 data cache access energy by an average of 37%, with no performance degradation.

Second, we discuss the explicit stack cache, where stack accesses are diverted to a separate stack cache. Because the footprint of the stack is small, this cache can be small and low-associativity, so accessing it takes less energy than for the data cache. We find that this saves approximately 36% of L1 data cache access energy on average, without significantly increasing static energy. In addition, the explicit stack cache can be virtually addressed and virtually tagged, eliminating the dynamic energy of address translation for 40% of L1 accesses, and it can be optimized for a larger percentage of writes by having a different write policy than the data cache. To keep this explicit stack cache coherent, physical coherence requests (e.g. from other cores) get translated back to virtual addresses using a small fully-associative buffer guaranteed to have the translation for any block in the stack cache.

The main contributions of this work are:

- An efficient method for classifying memory accesses as stack or non-stack that does not require compiler or operating system support
- An implicit stack cache with soft partitioning of the L1 data cache so that accesses to stack data require less dynamic energy
- An explicit stack cache which adds a separate L1 cache to store stack blocks, and which can be virtually addressed, virtually tagged to eliminate translation overheads, and which guarantees correctness in the presence of non-private stack data.

We expect this work can inspire future work leveraging other data characteristics to make memory hierarchies more energy efficient.

2. Background and Related Work

2.1 The Stack

The stack is one of the segments of virtual memory in a program. It stores return addresses and provides space for stack-allocated variables, such as local variables and values spilled from registers. It can also be used to pass parameters on function calls. It is a structure that normally grows on function calls and shrinks on returns. In x86, it is located at the top of a process's virtual address space, and grows downward towards the heap, as shown in Figure 1. It is contiguous in virtual space.

The stack pointer is a register that stores the virtual address of the top of the stack. Because function calls and references to local variables tend to be frequent, many accesses are made as offsets of the stack pointer. The stack

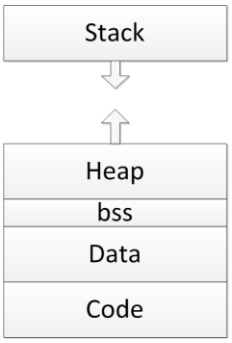

Figure 1. Layout of a program in virtual memory.

pointer is decremented on function calls and incremented on returns. The size of the stack region varies depending on the program and the operating system. However, it tends to be relatively small; in the latest version of the Linux kernel on x86, the default stack size is 8 MB, whereas the heap can be many gigabytes. In ARM Android, the stack is even smaller: 8 KB by default. If the stack pointer attempts to move beyond the allowed region, it results in a stack overflow error or segmentation fault.

2.2 Related Work

Previous work has suggested taking advantage of the differences between stack and non-stack data behavior to improve performance and energy efficiency.

Cho et al. [6] propose decoupling accesses to local variables, which are on the stack, from other memory accesses. They keep two separate data access queues, as well as separate caches for stack and non-stack accesses. Their approach was focused on improving performance of superscalar processors, and required compiler modifications.

Bekerman et al. [2] propose diverting memory accesses that use the stack pointer as a base to a separate stack cache. They find that for IA32, 99.5% of stack accesses are via the stack pointer, while 99.3% of non-stack accesses do not use the stack pointer. We show in Section 4 that this is not the case for x86_64 and ARM with our workloads.

Lee et al. [13] propose the Stack Value File (SVF), in which a non-architected register file is used to store the top of the stack. Their goal is to improve performance by avoiding accesses to the L1 data cache, both by diverting accesses to the SVF and eliminating writebacks of programmatically dead data to the data cache. Huang et al. [10] propose splitting the L1 data cache into stack and non-stack components in order to save energy. They also keep track of the top of the stack. These optimizations are based on the semantic properties of the stack, and rely on correct classification of stack and non-stack data. A snooping mechanism is used to redirect incorrectly classified accesses to the correct cache.

González-Alberquilla et al. [7] propose adding a filter cache before the L1 data cache that stores only the 4 to 64

words closest to the top of the stack. To determine whether an address should be stored in their filter cache, they do a subtraction from the stack pointer. This is on the critical path after address calculation and before the memory access.

Lee et al. [14] propose keeping separate structures for stack, heap, and global data. They rely on being able to distinguish these based solely on virtual address, possibly with some system support [15]. Without operating system and hardware support, these approaches will not work when Address Space Layout Randomization (ASLR) or Transparent Runtime Randomization (TRR) are used for security, because these techniques involve randomly relocating the program's stack, heap, and shared libraries at runtime [21].

Kang et al. [12] propose a virtually-addressed stack cache, to be used for the main application running on a server-type system. Their technique determines which accesses are stack by relying on both the stack pointer and the frame (base) pointer. This will not be effective on programs compiled with `-fomit-frame-pointer`, in which the frame pointer is used as a general purpose register. In addition, to indicate which process to cache in the filter cache, their scheme relies on operating system intervention, so their approach will not work on existing systems. They achieve performance benefits by having a "small-but-fast" stack cache, with a different L1 hit latency than the data cache, which can be difficult to deal with in the pipeline.

Much of the previous work ([6], [2], [13]) relies on determining whether an access is stack or not based on whether the address is an offset from the stack pointer register. We characterize the frequency of stack pointer accesses through the stack pointer in Section 4.

Another limitation of prior work is that the stack is assumed to be private. Although this is often the case, it is not guaranteed for x86 or ARM, and multithreaded programs may share data on the stack. Therefore, to execute existing programs correctly, the stack data must be kept coherent.

In addition, previous work relies on stack data having special semantics, such as a guarantee that data beyond the top of the stack will not be accessed [13] [6] [12] [7] [10]. This assumption is not necessarily safe, as we discuss in Section 3. In addition, it requires accurate classification of stack accesses for correct execution. Because guaranteeing that classification is always correct is difficult, we rely on a heuristic instead, but maintain correctness whether or not we misclassify some non-stack accesses as stack.

3. Characteristics of Stack Data

To optimize for stack data, it is first necessary to understand how common stack accesses are and how the characteristics of stack accesses differ from non-stack

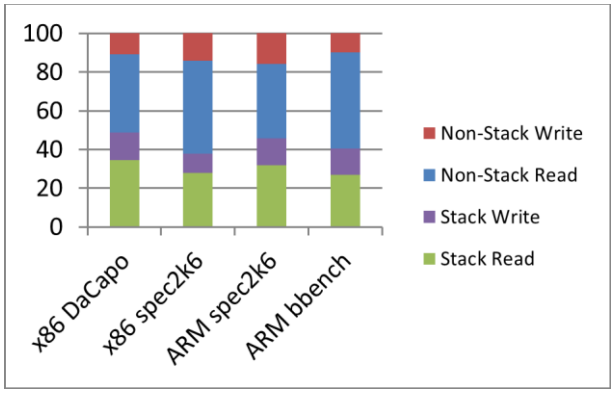

Figure 3. Breakdown of memory accesses for different workloads on x86 and ARM.

accesses. There are three characteristics we hypothesized that we could optimize for: (a) **frequent accesses**, (b) a **higher ratio of writes to reads** than for non-stack accesses, and (c) a **small footprint**.

To see if these characteristics held, we examined memory access patterns in a variety of applications. We used the gem5 simulator [3] in full system mode to characterize stack accesses for several benchmark suites for both x86_64 and ARM. We analyzed the workloads in SPEC 2006 [9] for x86_64 and ARM. We looked at DaCapo [4], which contains Java workloads, on x86_64. We also analyzed bbench for ARM Android; bbench is a webpage-rendering benchmark typical of what might be run on a phone [8] Detailed description of benchmarks can be found in Table 2.

Frequent accesses: We classified all memory accesses as either stack or non-stack for each of our workloads. Figure 3 shows that the percentage of stack for each benchmark suite on x86 and ARM. Figure 2 shows a breakdown for the SPEC 2006 workloads on x86, since these workloads showed the highest variation. On average, approximately 40% of the total accesses we observed were

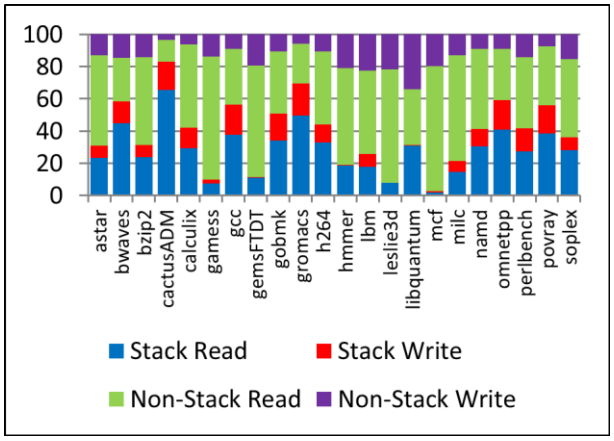

Figure 2. Breakdown of Memory Accesses by Workload for SPEC 2006 on x86

to stack data, although for DaCapo on x86 and SPEC on ARM, we saw close to 50% of accesses going to the stack. **Implication**: Stack is a large enough class of data cache accesses that optimizing specifically for it can have benefits for total energy efficiency.

Higher ratio of writes to reads compared to non-stack accesses: As shown in Figure 3, we find that on average, approximately 30% of stack accesses are writes, while only 22% of non-stack accesses are writes. Approximately half of total writes are to the stack. **Implication**: Having a different write policy for stack and non-stack data may have energy benefits by reducing L2 accesses or interconnect traffic.

Small footprint: We examined the size of the stack, both in terms of how much the stack pointer varies during the execution of a program, as well as how distant stack accesses typically are from the stack pointer, as shown in Figure 4. We find that the maximum change in stack pointer tends to be relatively small, even over the large intervals we measure.

We measured the distance between the stack pointer and each stack reference. We found that the majority (75%) of stack accesses are within 128 bytes, or two 64-byte blocks, on either side of the stack pointer, as shown by the CDF in Figure 4. In addition, 93% are within 1 KB of the stack pointer. Since most stack accesses are close to the stack pointer and the stack pointer remains within a small region, it follows that the footprint of the stack is small. In addition, the region it accesses tends to be contiguous, indicating that even a direct-mapped cache will not suffer from many conflict misses.

Implication: Even a small, low-associativity stack cache is likely to have a low miss rate.

Other characteristics of the stack: There are several other properties of the stack that are important to consider. One of these is whether the stack can be guaranteed to be private. Some previous approaches have assumed this to be true, but in x86 and ARM there is no guarantee that stack is private. For example, in a pthreads program threads can pass pointers to the stack. **Implication**: In order to guarantee correctness for multithreaded programs, it is necessary to keep stack data coherent between threads.

Another consideration is whether to use the semantic information about programmatic liveness provided by the stack pointer, which points to the top of the stack. Normally, the stack grows with function calls (pushes) and shrinks on returns (pops); anything beyond the top of the stack will be written before it is read again. In other words, any data beyond the top of the stack is programmatically dead, and need not be written back to any lower levels of the memory hierarchy.

Some previous approaches have taken advantage of this semantic information to avoid unnecessary writebacks. However, in Linux there is a "red zone" of 128 bytes past the end of the stack that is used for temporary storage [16]; any optimizations must not prematurely discard these blocks. In addition, multithreaded programs may have more than one stack, further complicating stack semantics. Finally, since we expect that stack blocks will not often be evicted from the cache due to their high locality, the benefit of avoiding writing back programmatically dead data will be limited. **Implication**: Optimizations based on the liveness of data beyond the stack are complicated and do not offer clear benefits.

In summary, stack data is **small**, **frequently accessed, and has many writes.** In addition, the stack is **not guaranteed private** and **determining liveness of stack blocks is not straightforward**. We find that these characteristics are true for both x86 and ARM. For these reasons, we advocate optimizing for stack accesses via stack caches, potentially with different write policies than for non-stack data. We do not make any assumptions about privacy or liveness, unlike prior work, and instead provide correctness in all cases.

4. Classifying Stack and Non-Stack Data

To treat stack and non-stack data differently, we first must have a way to distinguish them. We would like to make the distinction quickly and efficiently, as well as with high accuracy. Because misclassification is possible, we need to be able to handle it and make any necessary corrections, which may have a cost in performance and energy; for example, in some previous works such as SVF, misclassifications necessitated a pipeline flush [13]. We discuss several possible classification methods.

Compiler annotation: One approach is to have the compiler annotate each access as either stack or non-stack. However, the compiler may not be able to determine this

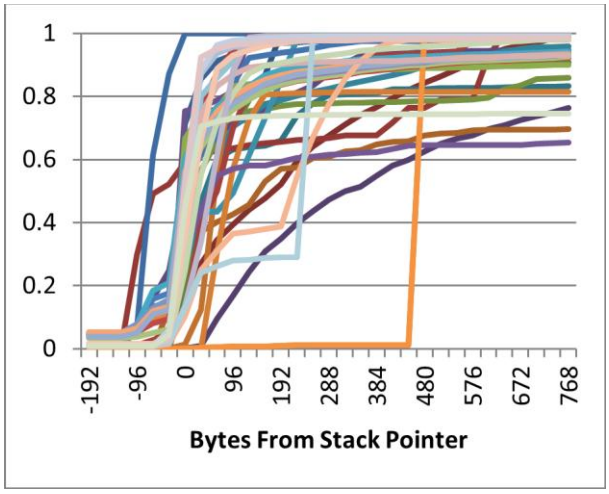

Figure 4. The cumulative distribution function for distance from the stack pointer for stack accesses for benchmarks in DaCapo and SPEC 2006 on x86 and SPEC on ARM.

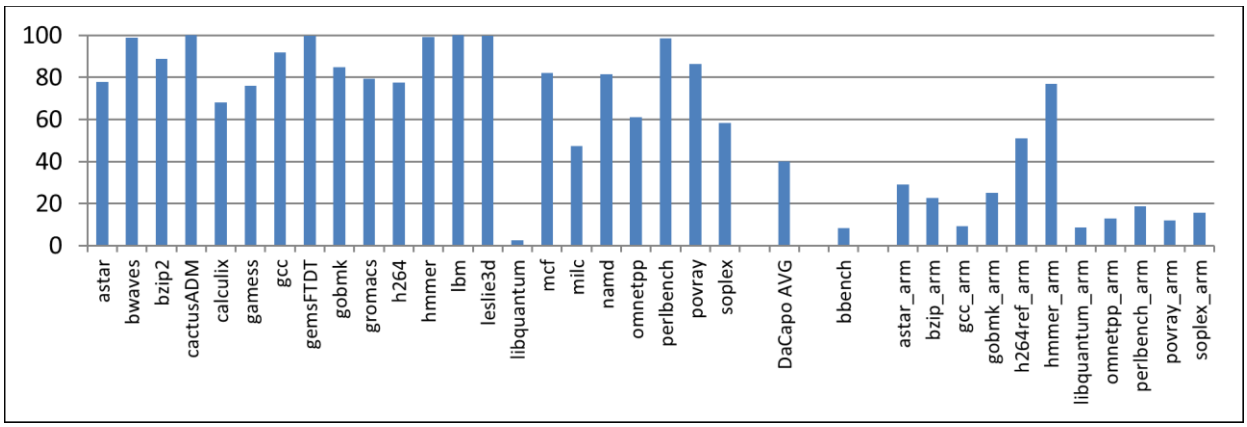

Figure 5. Percentage of accesses in the stack region that are relative to the stack pointer.

for all accesses that are made through pointers. In addition, this requires a change to both the compiler and the ISA, which we would like to avoid.

Offset from stack pointer: A second approach is to classify all accesses that are made relative to the stack pointer as stack, and all other accesses as non-stack. This approach has been used extensively in previous work. It allows classification in the decode stage, before the address has been calculated, which means that it does not add logic between address calculation and initiating the memory access. We looked at the percentage of stack accesses made via the stack pointer for several workloads, shown in Figure 5.

We find that on average, half of the accesses to the stack do not use the stack pointer, and in the case of libquantum on x86, 95% of stack accesses are not through the stack pointer and would be misclassified using this approach. Many of these accesses are likely through the frame (base) pointer; however, since it is possible to use that register as a general purpose register instead, classifying accesses via the frame pointer as stack would incur many misclassifications for programs compiled to not use the frame pointer. This high degree of inaccuracy would make it difficult to perform optimizations based on this classification scheme.

An alternate approach would be to mark which registers are pointers to the stack by keeping a bit associated with each one. This bit would always be on for the stack pointer, and would be set on registers when they were written as a result of a move or computation with a register with the stack bit set. Although this approach would likely be effective, it is ultimately more complicated than our chosen approach.

Virtual address of access: The approach used by Lee et al. [14] is to classify all accesses within a specific region of the virtual address space as stack. The stack is the area between the base of the stack and the stack pointer. However, this approach has several disadvantages. First, the base of the stack is not guaranteed to begin at any particular address, especially if Address Space Layout Randomization (ASLR) is being used. Second, subtracting the virtual address of the memory access from the stack pointer requires adding in a subtraction after the address has been calculated but before the memory access is initiated, which is on a cycle critical path. Third, it is unclear how this approach would work if there are multiple stacks in the same address space, possibly resulting in misclassifications.

Our choice: Aligned region of virtual address: An approach that is similar to the one above is to compare just the *N* most significant bits of the memory virtual address and the stack pointer. If these are the same, the access is classified as stack; otherwise, non-stack. This may classify "dead" blocks from beyond the top of the stack as stack, but as we are not using this liveness information, it will not affect correctness of execution. Because this approach uses the value of the stack pointer, which provides information about the virtual address where the stack is actually stored, it can work correctly in the presence of ASLR.

One advantage to this approach is that the classification can be done in parallel to address calculation: we can compare the most significant bits of the components of the address (base register, displacement, and index) with those of the stack pointer, and if and only if any of them match, it is very likely that the calculated virtual address matches as well. We found that this approach works well for user code, but x86 system code makes use of the segment register and is not accurately classified with this method. Because the percentage of time spent in system code is small, we choose to simply conservatively classify all accesses in system space as non-stack.

We compare all but the 23 least significant bits, for a stack region size of 8 MB for x86. This results in consistent classifications: as long as the stack pointer does not move between 8 MB regions, any address that was classified as stack in one access will also be classified as stack in the next, and vice versa. For ARM Android, we use an 8 KB

stack region. Our classification accuracies are discussed in section 7.3.

Importantly, since we do not rely on any special characteristics of the stack, such as programmatic liveness or privateness, misclassification will not have any effect on the correctness of our optimizations. However, since the optimizations are based on the differences in behavior that we have observed between the two classes, a high rate of misclassification may result in performance degradation.

5. Implicit Stack Cache

The *implicit stack cache* makes only minor changes to the data cache, but has significant energy benefits. We propose that blocks identified as stack be limited to residing in a particular cache way or ways, limiting the number of ways that need to be checked on a stack access.

In a set-associative cache, the tags need to be accessed, and if there is a match, the data from the corresponding way is used. Because it is important to keep L1 latency low, tags and data are usually accessed in parallel for the data cache. Although this reduces the latency, it results in an increase in dynamic energy because all data ways are accessed, even though at most one will contain the desired block.

One approach that aims to reduce the dynamic power without increasing latency in the common case is way prediction [11]. Only the way that is predicted to contain the data is accessed. If the tag does not match, the other ways can be checked. If the way prediction has high accuracy, this approach will reduce dynamic energy without impacting the L1 latency.

The implicit stack cache is a special case of way prediction: we predict that all stack accesses will be found in a particular cache way or ways. On a non-stack access, all ways are accessed in parallel, like in a typical data cache. On a stack access, we only access the way(s) where stack data is allowed to reside, thereby reducing dynamic energy. The advantage of our approach over a conventional way-prediction scheme is that we can make the prediction based on our classification, rather than performing a lookup in a PC or address-based prediction table.

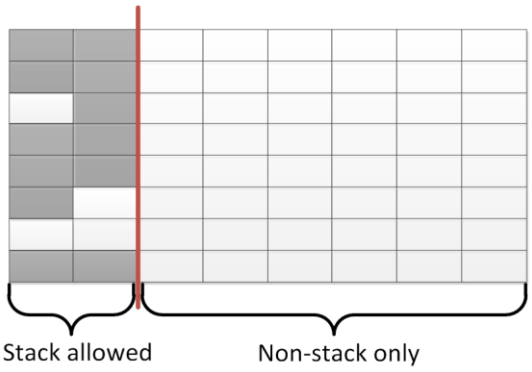

Figure 6. Implicit stack cache where stack data (dark) is allowed to occupy 2 of the 8 ways in the data cache.

To implement the implicit stack cache, we change the replacement policy so that on a miss, a stack block is only allowed to evict a block in one of the designated stack ways. Non-stack blocks can be inserted into the cache in any way, according to LRU or any other replacement policy. This results in a soft partitioning of the cache, where stack data is limited to a subset of ways, but non-stack data may reside in any way. Figure 6 shows a stack way cache where stack is only allowed to occupy the first two ways; blocks classified as stack are colored dark.

We can detect the case where a stack block is in one of the non-stack ways, such as due to a stack pointer change, by checking all tags on every access or checking all tags on a stack miss. To avoid the same stack address repeatedly missing because it is in a way designated as non-stack, the block can be evicted. When it is re-inserted into the cache on the next miss, it will be placed in an allowed way. This is the approach we take in our implementation.

As long as the stack classification is sufficiently accurate and the stack working set fits within the designated way(s), the performance impact will be small, and energy efficiency will increase because fewer ways are accessed on stack accesses.

As a concrete example, if the policy is LRU, the LRU information would be updated as normal on each cache access. When a block needs to be evicted to insert a non-stack block, the block in the LRU position should be chosen, regardless of whether it is in a stack way. It is only when the block to be inserted is classified as stack that there is any difference in the replacement policy: in that case, the least recently used of the blocks in the stack ways should be chosen. Because stack blocks tend to be frequently accessed, they should not often be evicted by non-stack blocks, and this soft partitioning scheme prevents the hit rate for non-stack blocks being penalized for applications with small stack footprints.

Although most L1 caches do not actually implement LRU, the implicit stack cache can be used with other replacement policies as well. For example, a common policy is pseudo-LRU, which can be implemented as a tree [18]. Then, each level of the tree is 1 bit, saying whether to go left or right to find the block to replace. The bits are also flipped as the tree is traversed, as well as updated on hits. This results in an approximation where the most recently used block(s) will never be replaced, but the exact order of replacement for the less recently used half of the line may not match LRU. Our approach works in this case as well; we can start the tree traversal directly at the node that is the earliest common ancestor of all stack blocks. In actual

systems, a lookup table for the pattern of bits is likely to be used rather than traversing the tree; we can simply add the bit pattern to find stack ways into this table.

6. Explicit Stack Cache

The *explicit stack cache* allows for several additional benefits over the implicit stack cache, at the expense of less flexibility in terms of adapting to varying stack footprints and more extensive changes from the common design. Rather than modifying the existing data cache, this design adds an additional cache to store stack data.

Accesses that are classified as stack are directed only to the stack cache, while all other accesses go only to the data cache. Then, on a miss, either the other cache can be checked or the L2 can maintain coherence by keeping a bit indicating whether the block is in one of the L1 caches.

The main advantage of this approach is that on hits, which are the common case, only one cache needs to be checked, and the stack cache can be much smaller and more energy-efficient than the data cache. The size of the stack cache can be chosen such that the hit rate is still high, but the cache is small. This is possible since the stack has a much smaller footprint compared to the entire working set.

Because the stack cache can be smaller and lower associativity than the data cache, it is possible for it to have a lower hit latency. However, allowing multiple L1 hit latencies complicates the pipeline. For our evaluation, we assume that the stack and data cache the have same latency.

6.1 Virtually Addressed, Virtually Tagged Cache

An additional optimization of an explicit stack cache over an implicit stack cache is that it can easily be virtually tagged. Because the TLB can be 3-13% of core energy [1], this will reduce energy without impacting performance.

On a stack cache hit, no translation is needed. In the case of a miss, translation is needed so that the block can be fetched from the next level of the cache hierarchy. During this step, the page permissions are checked.

There are several challenges when using virtually tagged caches [20]. One of these problems is *homonyms*: the same virtual address can map to different physical addresses. Homonyms can only occur when addresses from multiple address spaces are being stored in the same structure. Since the stack cache is private, this should only occur on context switches. To avoid this problem, we simply flush the stack cache on context switches. Since the stack cache is relatively small and context switches are infrequent, this will not have a significant impact on the hit rate of the stack cache.

One disadvantage of flushing the stack cache on a context switch is that it can increase the context switch latency. To avoid this, the blocks in the stack cache can be marked as "stale" on a cache flush, indicating that they are no longer valid to write to but are dirty. Then, on a later "hit" to a stale block, the block can be written back to the L2 and replaced with the block that the virtual address refers to. This is similar to the swapped-valid bit used by Wang et al [20]. This prevents having to flush the entire stack cache at once, while still providing correct behavior.

Table 1. Simulated environment.

L1 Instruction Cache	32KB, 2-way, private, write-back, inclusive to L2
L1 Data Cache	32KB, 8-way, private, write-back. inclusive to L2
L2 Cache	512KB, 16-way, shared
Processor	In-Order, Simple Timing
Number of cores	1

If the core is multi-threaded, we could add a thread ID to each line of the stack cache or provide each thread with its own stack cache.

Another problem with a virtually tagged cache is *synonyms*: multiple virtual addresses mapping to the same physical address. We do not expect this to be a common occurrence for blocks in the stack cache, because generally the stack is a contiguous region. However, we must guarantee correctness even if a program is doing unusual things with the stack or if data is misclassified. In addition, it is necessary to keep the stack cache coherent with both the other stack caches and data caches in the system; it is possible that one processor might classify an address as being stack and another as non-stack.

To this end, we propose keeping a small buffer containing virtual to physical translations along with the stack cache. Because most entries in the stack cache are expected to come from only a small number of pages, this structure can be kept small. We will evaluate the size this structure would need to be in section 7.4. It can function both as the TLB for the stack cache, allowing virtual to physical address translation on a writeback from the stack cache, as well as to determine whether it is possible that an incoming coherence request is for a block residing in the stack cache. If the physical page of a cache block is not found in this structure, then the block is not in the stack cache, eliminating any need to check it. Because the structure is small, checking it on coherence requests will not take much energy.

6.2 Writeback Policy

A second optimization that can be made for the stack cache is to make it writeback, even if the data cache is writethrough. One reason why the data cache might be writethrough is that the ECC information can be kept within the L2, and the data cache can use weaker, but less expensive, parity. Then, if an error is detected in the data cache, the block can be invalidated and re-fetched from the L2, which is guaranteed to have a valid and up-to-date copy. This approach reduces the energy and area overhead of the data cache, at the expense of requiring writing two

Table 2. Benchmarks

Benchmark Suite	Benchmarks
SPEC 2006 for x86	astar, bwaves, bzip2, cactusADM, calculix, games, gcc, gemsFTDT, gobmk, gromacs, h264, hmmer, lbm, leslie3d, libquantum, mcf, milc, namd, omnetpp, perlbench, povray, soplex
DaCapo for x86	avrora, batik, eclipse, fop, h2, jython, luindex, lusearch, pmd, sunflow, tomcat, tradebeans, tradesoap, xalan
Bbench for ARM	bbench
SPEC 2006 for ARM	astar, bzip2, gcc, gobmk, h264, hmmer, libquantum, omnetpp, perlbench, povray, soplex

structures on every store. It is used in AMD's BullDozer [5].

Because stack blocks are written very frequently, having a writethrough policy for these blocks will incur a large number of extra writes to the L2 (or to the write coalescing cache, in Bulldozer). We can instead use a writeback policy for the stack cache, and protect it with ECC. Since the stack cache can be significantly smaller than the L1 data cache while still maintaining a high hit rate, the overhead of ECC can be more easily accommodated.

7. Evaluation

7.1 Methodology

We evaluated both the implicit and explicit stack cache configurations on the gem5 simulator, using Ruby as the memory model [3]. We ran all x86_64 workloads in full system mode, under Linux kernel 2.6.22.9. For ARM, we used Android Gingerbread for bbench and Linux kernel 2.6.38.8 for the other workloads. We used CACTI 6.5 to model the relative static and dynamic energy of each configuration [17]. We report relative energy numbers because we believe that CACTI is more reliable for relative numbers than absolute. We obtained energy estimates for 32 nm technology with type itrs-hp and fast access mode.

7.2 Benchmarks

We ran a selection of the SPEC CPU2006 [9] workloads for x86_64 and ARM, as well as DaCapo [4] for x86_64 and bbench [8] for ARM. We list the selection of benchmarks we evaluated in Table 2.

7.3 Implicit Stack Cache

We simulated a 32 KB, 8-way set-associative data cache with stack data allowed to reside in numbers of ways varying from 1 to all 8. The case with 8 ways is the baseline case, which behaves exactly the same as a normal data cache, and we normalized execution times and performance to it.

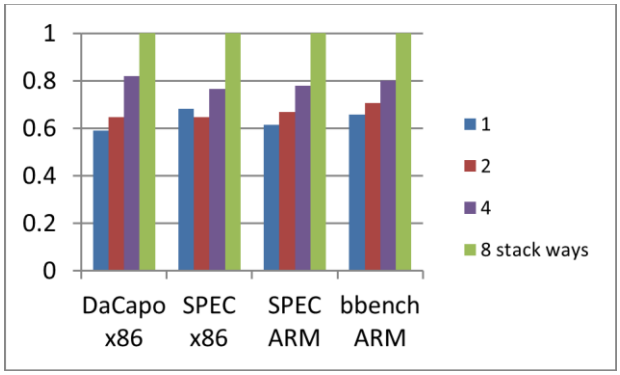

Figure 7. Dynamic cache energy for implicit stack cache, when stack is allowed to reside in 1, 2, 4 or all ways, normalized.

We find that this approach significantly decreases overall energy. As shown in Figure 7, limiting stack data to only one way results in a reduction of 37% of dynamic L1 data cache energy, on average. Limiting the stack to two ways also reduced energy, but by slightly less: on average, 30%. The benefits decrease as stack is allowed to reside in more ways.

In order to reap the maximum benefits of the implicit stack cache, we would like to allow stack to reside in the minimum number of ways that can hold the entire stack working set. This allows saving energy without increasing the miss rate and hence causing performance degradation. To get an idea of how many ways are required to hold the working set of the stack in each of our workloads, we measured the misses per thousand instructions broken down into stack and non-stack misses for each configuration, as shown in Figure 8. Most workloads do not show an increase in miss rate when stack is limited to fewer ways, implying that performance will not be affected.

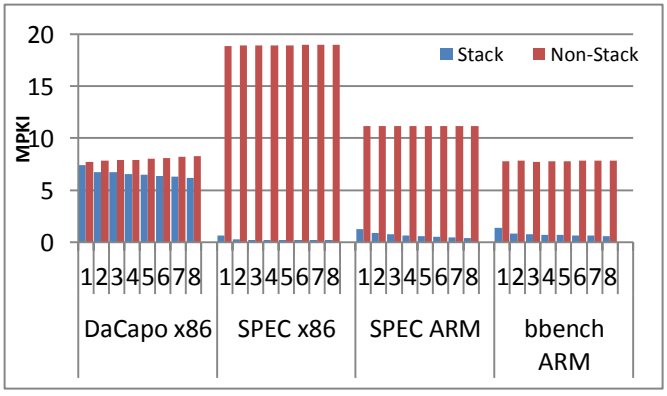

Figure 8. Misses per thousand instructions for implicit stack cache, where 2 indicates stack is limited to 2 ways.

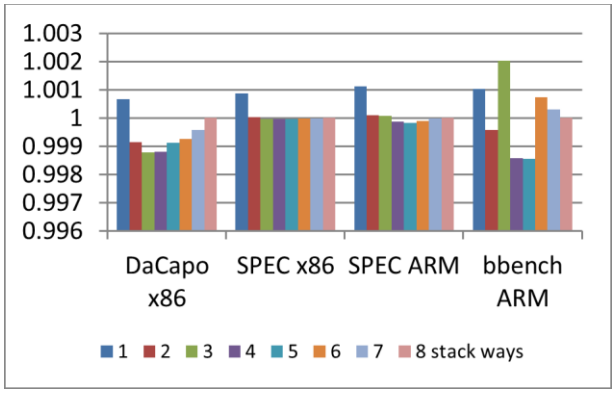

Figure 9. Execution time for implicit stack cache, normalized.

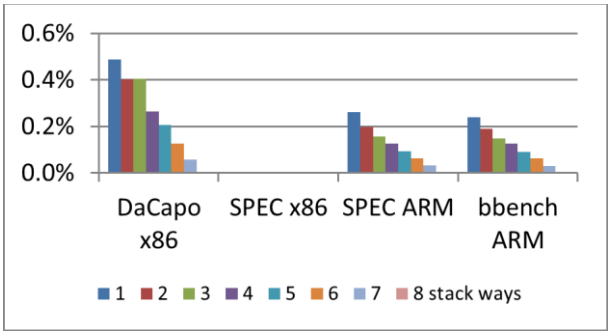

Figure 10. Percentage of stack accesses found in non-allowed ways.

Indeed, as shown in Figure 9, we find that performance is generally not affected when the stack is confined to a subset of ways; even with the stack confined to 1 way, the overhead is less than 1%. For 2 ways, there is a slight speedup for some workloads (such as DaCapo). This is possible because it prevents stale stack data from polluting the cache.

To test the accuracy of the stack classification, we looked at the percentage of stack accesses where the block was found in a non-allowed way. Our results are shown in Figure 10, and show that a maximum of 1.2% of stack accesses are found in a non-allowed way. This confirms that this case is uncommon. Therefore, having extra latency in these cases or causing an extra L2 access will not have a large effect on energy or performance, and our classification approach works well enough that it is unlikely a more accurate strategy would offer any significant improvement.

7.4 Explicit Stack Cache

The explicit stack cache targets energy improvements in three ways: accessing a smaller structure rather than the data cache, avoiding address translation, and optimizing the write policy. We find that all three of these have significant benefits.

We simulated with an explicit stack cache with sizes from 1 KB to 16 KB and associativity from 1 to 4. For our initial experiments, we assumed both caches were write-back and physically tagged. We shrunk the data cache to 28 KB, 24 KB, or 16 KB, so that the total L1 data capacity remained at most 32 KB. The static power for the two caches remained approximately constant, since cache capacity did not change. For our dynamic energy estimates, we conservatively assumed that the data cache always took a constant energy per access; that is, we assumed that the 16 KB data cache had the same energy per access as the 32 KB. This was done due to limitations in CACTI.

Accessing a smaller structure:

On an access classified as stack, we first check the stack cache. If the block is found there, there is no need to check the data cache. Because the stack cache is much smaller than the data cache, accessing it instead will take less energy. For non-stack accesses, we check only the data cache. On a miss to either cache, we could either check the other cache and then the L2, or check only the L2 and rely on coherence mechanisms.

We find that the average reduction in L1 data dynamic

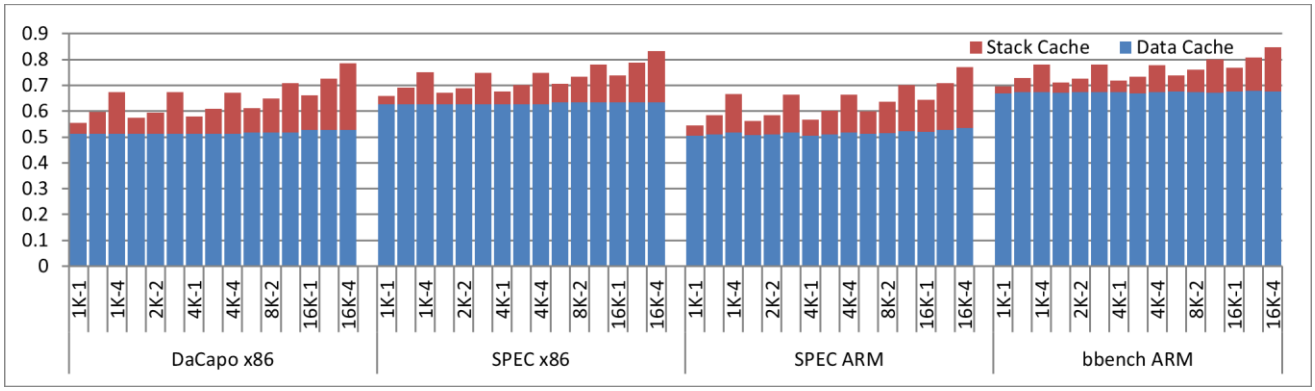

Figure 11. Dynamic energy for explicit stack cache + data cache configurations, normalized to combined data cache. 2K-1 indicates a 2 KB, direct-mapped stack cache.

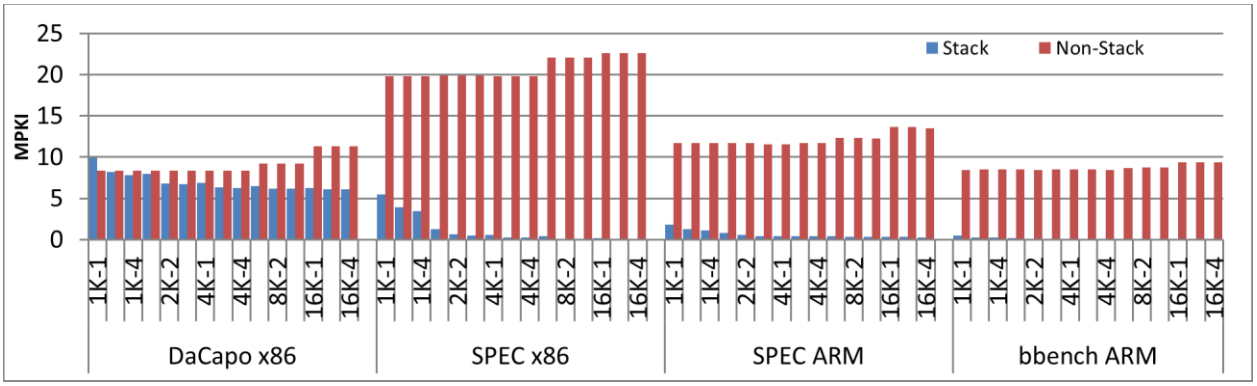

Figure 12. Misses per thousand instructions for the explicit stack cache.

energy for the most energy efficient configuration is approximately 36% for both ARM and x86, as shown in Figure 11. We break down the dynamic energy into the components for stack and data cache. The maximum energy reduction we saw was approximately 73% for CactusADM on x86 for a direct-mapped, 1 KB stack cache, because CactusADM has a small stack footprint. All configurations we examined reduce dynamic energy on average. Although increasing the associativity for the stack cache can increase the hit rate, it can also increase the dynamic energy because each stack cache access takes more energy. We show the number of misses per million instructions for each configuration in Figure 12, and the normalized execution time in Figure 13. We see that adding a stack cache and reducing the size of the data cache adds a slight performance overhead due to an increase in data cache misses, but that it is generally less than 1%.

The exact design point can be selected based on the desired trade-off between performance and energy. If the stack cache is too small, the stack miss rate will increase; similarly, if the data cache is too small, the data miss rate will increase. We see that for a 4 KB, direct-mapped stack cache, the dynamic energy is reduced by about 36% for the

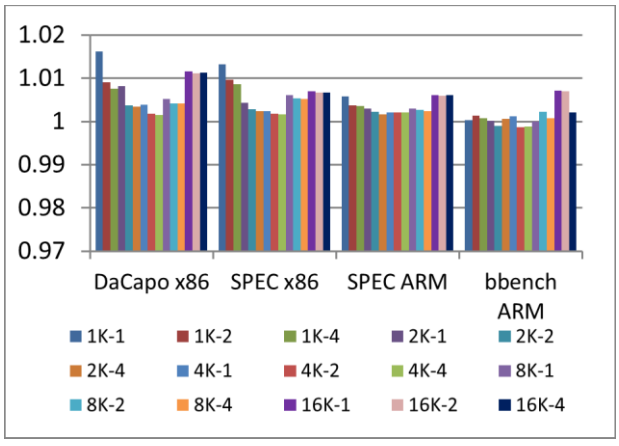

Figure 13. Normalized execution time for explicit stack cache.

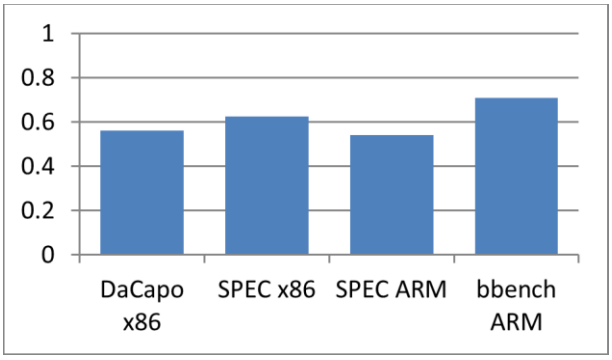

Figure 14. TLB accesses for a virtually-tagged stack cache and physically-tagged data cache (normalized).

data cache with approximately 0.2% performance overhead; this is approximately the same increase in energy efficiency as for the implicit stack cache. However, there are several other benefits associated with an explicit stack cache that we will now discuss.

Avoiding address translation:

We evaluated the potential energy savings from having a virtually addressed, virtually tagged stack cache, and found that on average, the number of translations can be reduced by 40%. Our results are shown in Figure 14. This assumes that translation occurs on every non-stack L1 access, and on every stack cache miss. Because TLB energy is 3 to 13% of core energy, the benefit of avoiding 40% of the accesses and thus up to 40% of the dynamic energy for address translation is significant.

As mentioned in Section 6.1, the virtually indexed virtually tagged stack cache must be kept coherent, and coherence requests are by physical address. If the virtual page to physical page translations are kept in a small buffer, the lookup can be done in either direction without a large energy or latency. We looked at how many different pages are represented in the stack cache for each workload. The maximum number of pages ever simultaneously found in the stack cache is given in Figure 15. For most x86 SPEC workloads, the maximum number of pages is less

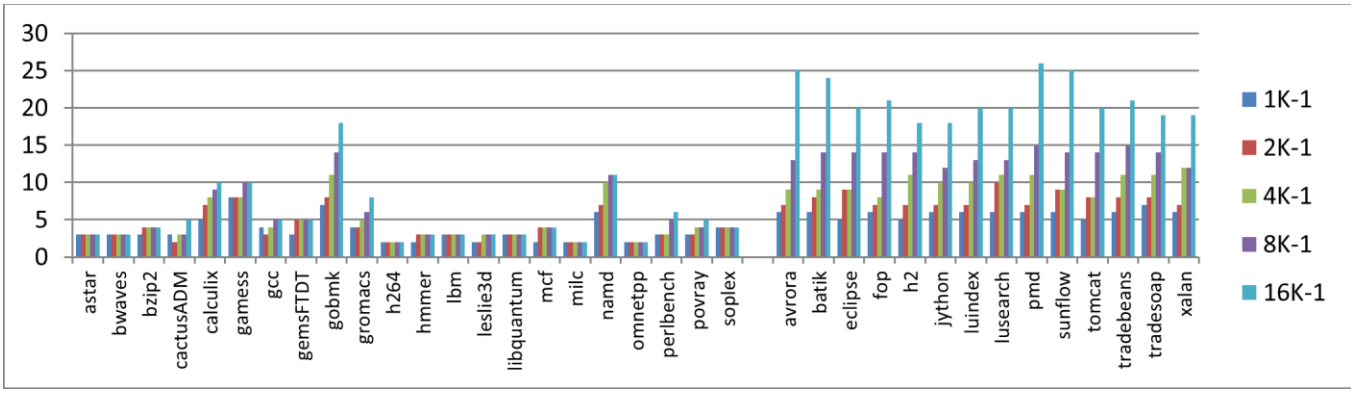

Figure 15. Maximum number of pages in stack cache at a time.

than 5. It is higher for DaCapo, but for a 4 KB cache, it is usually less than 10. This indicates that a small, fully-associative buffer that is only checked on coherence requests (generated when some cache has a miss) will be sufficient to keep the stack cache coherent without large energy overheads.

Optimizing the write policy:

We evaluated potential benefits of having a writeback stack cache in the case where the data cache is writethrough and found that on average, this strategy allows us to eliminate 43% of the L2 accesses when the stack cache is 4 KB. Our results are shown in Figure 16. In processors that have a writethrough data cache, there may be a small write coalescing cache between the data cache and the L2, to reduce the frequency of L2 accesses. However, every write still must access both the data cache and the write coalescing cache. Our approach results in only accessing one small structure on every stack write. For benchmarks like gromacs on x86, where the majority of writes are to the stack, this can eliminate a large number of unnecessary writes to the write coalescing cache or L2 (up to 75%).

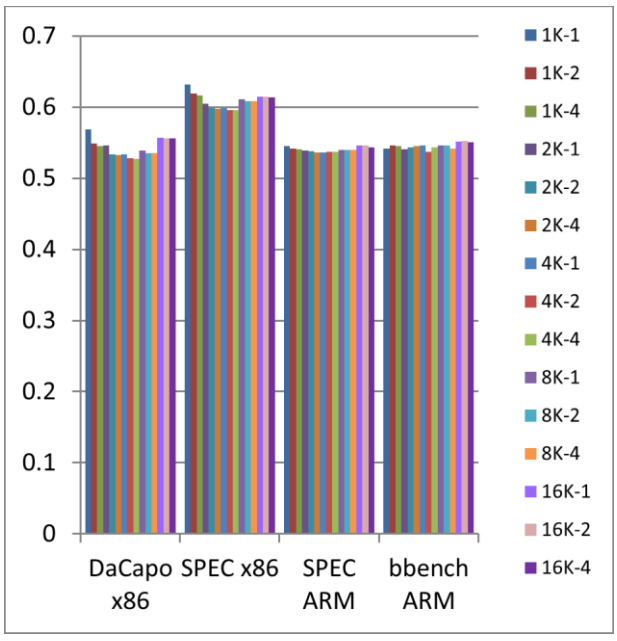

Figure 16. Number of L2 accesses with writeback stack and writethrough data cache (normalized).

7.5 Comparison of Implicit and Explicit Stack Caches

Both approaches have advantages and disadvantages. The implicit stack cache can easily adapt to workloads with unusually small stack footprints, because the cache is only soft partitioned and blocks not occupied with stack can be used by non-stack data. However, the implicit stack cache does not allow optimizations such as avoiding address translation or unnecessary write-throughs of stack data.

8. Conclusion

Energy efficiency is an increasingly important consideration across a range of processor types, from embedded processors through servers. Where there are ways to improve energy efficiency without negatively affecting performance, doing so is well worth it. We discussed one method of doing so: taking advantage of the differing characteristics of stack and non-stack data. Our approach requires no changes to user or system code and does not make any assumptions about stack semantics. We showed that by storing stack accesses either in a designated way of the L1 data cache or in a separate structure parallel to the data cache, we can achieve significant energy savings: 37% of data cache dynamic energy for the implicit stack cache and 36% for the explicit stack cache, with additional savings of 40% of address translations. These energy benefits are obtained without sacrificing any performance or requiring any changes to existing code or systems.

9. REFERENCES

1. Basu, A., Hill, M.D., and Swift, M.M. Reducing Memory Reference Energy With Opportunistic Virtual Caching. *Proceedings of the 39th annual international symposium on Computer architecture*, (2012), 297–308.
2. Bekerman, M., Yoaz, A., Gabbay, F., Jourdan, S., Kalaev, M., and Ronen, R. Early load address resolution via register tracking.

Proceedings of the 27th annual international symposium on Computer architecture, ACM (2000), 306–315.

3. Binkert, N., Beckmann, B., Black, G., et al. The gem5 simulator. *Computer Architecture News (CAN)*, (2011).
4. Blackburn, S.M., Garner, R., Hoffman, C., et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, ACM Press (2006), 169–190.
5. Butler, M., Barnes, L., Sharma, D.D., and Gelinas, B. Bulldozer: An Approach to Multithreaded Compute Performance. *IEEE Micro 31*, 2 (2011).
6. Cho, S., Yew, P.-C., and Lee, G. Decoupling local variable accesses in a wide-issue superscalar processor. *Proceedings of the 26th annual international symposium on Computer architecture*, IEEE Computer Society (1999), 100–110.
7. Gonzalez-Alberquilla, R., Castro, F., Pinuel, L., and Tirado, F. Stack oriented data cache filtering. *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ACM (2009), 257–266.
8. Gutierrez, A., Dreslinski, Wenisch, et al. Full-System Analysis and Characterization of Interactive Smartphone Applications. *the proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*, (2011), 81–90.
9. Henning, J.L. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News 34*, 4 (2006), 1–17.
10. Huang, M., Renau, J., Yoo, S.-M., and Torrellas, J. L1 data cache decomposition for energy efficiency. *Proceedings of the 2001 international symposium on Low power electronics and design*, ACM (2001), 10–15.
11. Inoue, K., Ishihara, T., and Murakami, K. Way-predicting set-associative cache for high performance and low energy consumption. *Proceedings of the 1999 international symposium on Low power electronics and design*, ACM (1999), 273–275.
12. Kang, S. chan, Nicopoulos, C., Lee, H., and Kim, J. A High-Performance and Energy-Efficient Virtually Tagged Stack Cache Architecture for Multi-core Environments. *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, (2011), 58 –67.
13. Lee, H.-H.S., Smelyanskiy, M., Tyson, G.S., and Newburn, C.J. Stack Value File: Custom Microarchitecture for the Stack. *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, IEEE Computer Society (2001), 5–.
14. Lee, H.-H.S. and Tyson, G.S. Region-based caching: an energy-delay efficient memory architecture for embedded processors. *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, ACM (2000), 120–127.
15. Lee. Improving Energy and Performance of Data Cache Architectures by Exploiting Memory Reference Characteristics. 2001.
16. Matz, M., Hubicka, J., Jaeger, A., and Mitchell, M. System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99. *Available at www.x86-64.org/documentation/abi.pdf*, (2009).
17. Muralimanohar, N., Balasubramonian, R., and Jouppi, N.P. *CACTI 6.0*. Hewlett Packard Labs, 2009.
18. So, K. and Rechtschaffen, R.N. Cache Operations by MRU Change. *IEEE Transactions on Computers 37*, 6 (1988), 700–709.
19. Sodani, A. *Race to Exascale: Opportunities and Challenges*. 2011.
20. Wang, W.H., Baer, J.-L., and Levy, and H.M. Organization and performance of a two-level virtual-real cache hierarchy. *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, ACM (1989).
21. Xu, J., Kalbarczyk, Z., and Iyer. Transparent runtime randomization for security. *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, (2003), 260 – 269.