Implementing a Stack Cache

Alex Hemsath, Robert Morton, Jan Sjodin
Rice University: Elec525 – Advanced Microprocessor Architecture

Abstract – Modern architectures provide mechanisms to allow a process to manipulate a portion of its memory as a stack through special registers and instructions. If the portion of memory that represents the stack were kept in a cache hierarchy separate from the traditional data cache, the so-called "stack cache" could be tuned to the specialized behavior of stack access patterns. We have simulated a simple stack cache within the SimpleScalar framework. For four of the SimpleScalar benchmark applications the stack cache we implemented produced speedups in the range of 1–4%.

1. Introduction

Memory access patterns for stack data have both spatial locality and temporal locality since the data occupies a small, contiguous area of memory near the top of the stack. The stack is heavily used in modern programs compiled from high-level languages for bookkeeping and parameter passing during function calls, allocating local variables and register spilling.

This paper focuses on implementing a stack cache to handle the specific nature of stack access patterns and alleviate pressure on the L1 data cache.

1.2 Prior Work

Cooper and Harvey identified an interesting problem in the traditional assumption that compilers can do very little in the way of understanding run-time behavior of memory accesses[1]. This assumption ignores compiler inserted spills and fills from register allocation, parameter passing and procedure calls, all which observe the temporal locality of data with respect to the top of the stack. Their approach did not convince most readers that a "compiler controlled memory" would help solve this problem. Rather than repeat their experiments to verify this, we propose a purely architectural solution.

1.3 Stack Cache Basics

The stack cache acts as a window into the current stack memory of a program, containing the data that has most recently been used. Specifically it will contain all data within a certain offset from the top of the stack. As the stack grows, old data at the bottom of the stack cache must be evicted to make room. The stack cache can be viewed as a circular buffer with an index for the top of the stack and one for the bottom of the stack:

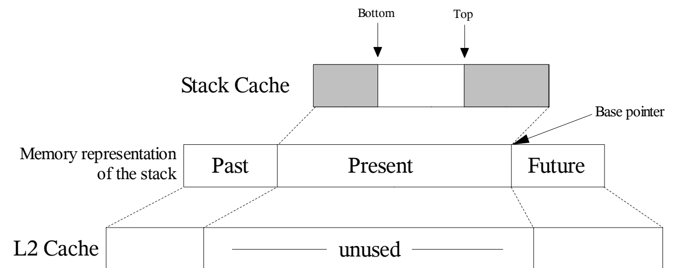

Figure 1: A simple stack cache diagram

Older entries in the stack can be preemptively saved to the L2 cache as the stack cache gets full, predicting the necessity of spilling. When the stack shrinks, data will need to be loaded back into the cache from L2 or main memory, and again this can be done preemptively - data can be prefetched into the stack cache before it is known that it is needed. This preemptive spilling and filling of the stack cache could greatly reduce latency of accesses to stack data by making sure the data is always available in the stack cache – assuming the program is accessing the stack data according to the rules of stack temporal locality

(i.e. only a small window of the stack is in active use at a given time).

1.4 Paper Overview

Section 2 covers our motivation and hypotheses in depth. The details of the proposed hardware implementation are presented in Section 3, while the software implementation and simulator details are discussed in Section 4. We analyze the simulation results in Section 5, and draw our conclusion in Section 6.

2. Motivation and Hypotheses

In the previous sections we mentioned the possible usefulness of a cache for a particular type of memory access. In this section we go into more detail, including why current cache designs are ill-equipped to handle the temporal and spatial locality exhibited by stack accesses.

2.1 Motivation

A large amount of research in computer architecture of the past fifteen years has gone into overcoming the CPU-memory bottleneck, which not only increases latency of individual instructions, but limits instruction level parallelism (ILP) through long-latency chains of data dependencies. The standard architectural element used to manage the increasing gap between CPU cycle times and memory access latencies is to cache a small subset of main memory within a fast memory more local to the CPU.

However, many characteristics of memory use limit the extent to which locality can be exploited – conflict misses in particular are a limiting factor. One potential source of such misses is the conflict between stack data and heap data, which are, typically, logically separate areas of a program's memory space. Also, cache sizes and associativity have increased over the years in the attempt to decrease miss rates and improve performance, but this comes at the expense of increasing cache latency and increasing power consumption. Originally, L1 data caches were designed to have single-cycle access latencies, but in typical state-of-the-art processors latencies of 3 cycles are more common.

L1 data caches have become all-purpose caches, catering to random-access data patterns more than patterns with structure. Because of this they perform poorly with an even mix of stack and heap accesses.

2.2 Hypotheses

There are multiple benefits that could be seen from the stack cache, which we verify independently in Section 5. The following are our hypotheses:

1. The stack cache will have a faster access than modern L1 caches. The stack cache maps a contiguous window of memory representing the top of the stack, so it can be implemented as a fast direct-mapped cache.

2. It will keep register spills and fills and other short-lived stack (memory) operations from evicting useful longer-term heap data from the L1 cache.

3. The stack cache will be to greatly decrease the latency of accesses to stack data because of the more predictable nature of stack accesses. The preemptive filling and spilling mentioned in Section 1.3 will attempt to keep the "current" stack data in the stack cache only when it is need, thus reducing latency to stack data for all stack accesses. This reduced latency in turn will increase ILP by relieving data dependency pressure on the instruction stream.

We hypothesize that these concepts operating in concert will improve performance mostly for user code, but possibly for scientific code as well.

One important consideration in designing new architectural elements is that they integrate well with existing designs, scale well for the

future, and require a minimum of compiler support (if any) to be taken full advantage of. We have designed our stack cache to complement existing cache structures rather than modify or replace them. Additionally we will see that the stack cache is a small, simple structure, which allows it to scale well. Since the stack is a run-time memory structure, no compiler support is anticipated to be necessary.

3. Architectural Details

The stack cache uses two pointers named Top and Bottom; they keep track of the top of the stack, and the bottom of the valid cached stack data, respectively. Data in the stack cache between the Bottom and Top pointers represents valid data that is coherent with the L2 cache. The entire cache structure maintains an array of the stack data as well as a matching array of dirty bits to mark changes to stack data. Our policy for saving data to the L2 cache is write-back.

The stack cache can be implemented as a simple direct mapped cache since it maps to a contiguous window of memory that represents the top of the stack. Because of this, access times can be limited to one cycle unlike other modern cache designs. The stack cache is a circular buffer, allowing large increases in stack size to wrap around in the cache.

The stack cache unit needs to monitor updates to the stack pointer, which is sometimes ABI (Application Binary Interface) specific, OS-specific, or architecture-specific. We assume for our study that the pointer to the top of the stack is always maintained in the same register. When the stack pointer is modified, the Top and Bottom pointers will need to be adjusted to reflect the new window of valid data.

The stack cache prefill/spill unit should look ahead in the instruction window to determine if the stack is likely to grow or shrink in the near future. Using this information it can determine when to transfer data between the stack cache and the L2 cache in advance to prepare the stack cache for the sudden change in its data window. This is the feature that allows low latency stack accesses for data that would normally have missed the stack cache.

Specifically, when the system is using the L2 access bus relatively infrequently the stack cache prefill/spill unit can begin the write-back process for data that is marked dirty. This allows the data at the bottom or top of the current stack window to be evicted from the stack cache quickly in the event of a large increase or decrease in stack size. However until the stack window is modified the data will remain accessible and valid.

4. Experimental Software Implementation

We used the SimpleScalar 3.0 framework for implementing our design, given its thorough support of SPEC benchmarks and important basic architectural features. Specifically we designed our stack cache into a system that has two levels of cache and an out-of-order core.

4.1 Preliminary Analysis

We used sim-cache to examine stack access behavior of several SPEC benchmarks. The statistics verified our belief that a significant portion of the memory accesses were to the stack (34-60% among the benchmarks we ran). Additionally we were able to measure how large the stack cache would have to be to effectively cover the range of stack accesses in a typical program. The following data shows stack access patterns for the *equake* benchmark:

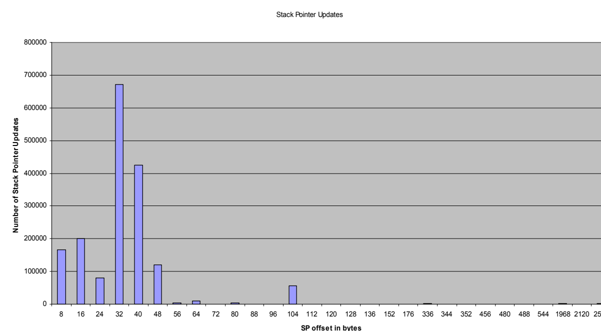

Figure 2: Stack Pointer modification in bytes.

The same tests with other benchmarks indicate similar access patterns. Since the stack is accessed mostly in increments less than 2Kb, it is reasonable to select a stack cache of approximately the same size. We tested the stack cache in sizes of 2Kb and 4Kb to get a good measure of the size requirement.

Additionally we discovered that the Stack Pointer is usually modified once every 44 to 141 instructions, which is ample time for the prefill/spill unit to aggressively predict modifications to the stack window. These modifications to the stack pointer are mostly less than 400 bytes, which means that the prefill/spill only needs to preemptively transfer data to cover a small percentage of the stack cache size.

4.2 Implementation Simplifications

We chose to implement the stack cache exclusive from the L1 cache: all accesses to memory locations that are offset from the stack pointer would be directed to the stack cache and never access the L1. This occurs even when the access would incur a miss in the stack cache. This is done to simplify the coherency issues that would occur if we allowed the same data to reside in both the L1 and the stack cache. All stack data resides in the stack cache or L2 cache, or in main memory.

We additionally put limitations on our spill/fill strategy, making it conservative to avoid complicating our analysis in dealing with a saturated L2 bus. Instead of looking ahead in the instruction stream, it maintains a view of the valid portion of the stack cache at every cycle. When the stack cache contains less than 50% valid data, it begins to fill the stack cache from the L2 during periods of low usage of L2 bandwidth.

When the stack cache contains more than 50% valid data it begins spilling to L2, writing data that is marked as dirty and then resetting each dirty bit. If the Stack Pointer is modified in an increment so large it invalidates the entire stack window, the stack cache must stall all access until the stack cache is made coherent with the L2 cache. Additionally the L2 access bus is locked by the stack cache so it has highest priority to complete the stack data updates.

Our implementation took approximately 500 lines of code, plus some datalogging code. All of our additions were to sim-outorder, which is the out-of-order portion of SimpleScalar.

4.3 Simulated Hardware Configurations

Each configuration uses some common hardware settings. We assume an architecture with a 512Kb, 8-way set-associative L2 cache (shared between data and instructions). The latency of the L2 is 12 cycles, and the latency between the L2 and main memory is 80 cycles (assuming modern access times for main memory, and a 1 GHz processor). The L1 instruction cache is always 16Kb, 4-way set-associative.

Our baseline test is an architecture with no stack cache and a 16Kb, 4-way L1 data cache. The baseline latency for the L1 data and instruction caches is 3 cycles. Our other configurations consisted of varying the L1 data cache size, the L1 latency, the L1 associativity, and the stack cache size. Each other configuration will be fully described as it is introduced.

5. Simulation Results and Analysis

We present the data as charts comparing how the different configurations behave and perform. We had a number of surprising results, but we feel the data verifies our hypotheses.

5.1 Varying Stack Cache Size

Preliminary analysis showed that most stack accesses occur within 2Kb of the Stack Pointer, so we compared a 2Kb stack cache and a 4Kb stack cache to the baseline configuration. In each configuration the stack cache had an access latency of 1 cycle, reflecting the fast lookup mechanism.

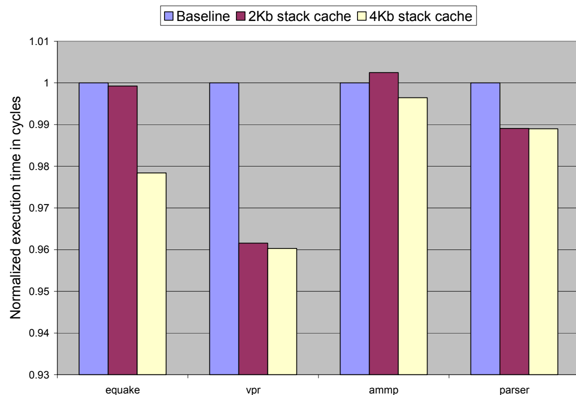

Figure 3: Varying stack cache size

It is clear from these results that the stack cache has somewhat varying effects on overall performance. In some cases a 2KB stack cache performs as well as a 4KB stack cache, and in other cases the doubling in size of the stack cache shows an improvement. The two benchmarks in which increasing the size of the stack cache does show an improvement are from SpecFP which is consistent with our data for the stack access patterns of floating point versus integer code.

However the large performance difference between 2Kb and 4Kb stack caches is anomalous considering our preliminary statistics that shows most data fitting within a 2Kb cache. This makes sense when one considers our prefill/spill policy – we essentially only maintain the stack cache at 50% utilized for the majority of execution. Thus a 4Kb cache behaves as a 2Kb should if it had an effective prefill/spill unit.

5.2 Stack Cache / L1 Cache miss rates

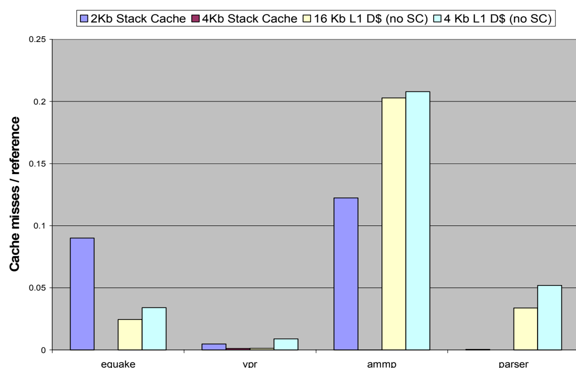

Figure 4: Cache miss rates

We measured the miss rates of the 2Kb and 4Kb stack cache configurations and compared them to the miss rates of a 16Kb L1 data cache and a 4Kb L1 data cache. The results are shown in Figure 4. The two floating point benchmarks, *equake* and *ammp*, showed the greatest decrease in cache misses between the 2Kb stack cache and the 4Kb stack cache. This correlates strongly with the increase that both these benchmarks see in execution speed in going from a 2Kb to 4Kb stack cache.

The miss rate for the 16Kb and 4Kb L1 data caches were very similar for each benchmark, indicating that programs have a hard time taking full advantage of a larger L1. Because of the memory bottleneck however, each extra cache hit helps and justifies the traditionally larger L1 size. The 4Kb stack cache outperforms the 16Kb L1 cache in reducing cache misses, which indicates that the stack cache is an extremely effective structure.

5.3 L1 cache access counts

This data shows how many memory accesses we are diverting from the L1 into the stack cache. Specifically it's a measure of how many stack accesses there are relative to heap accesses.

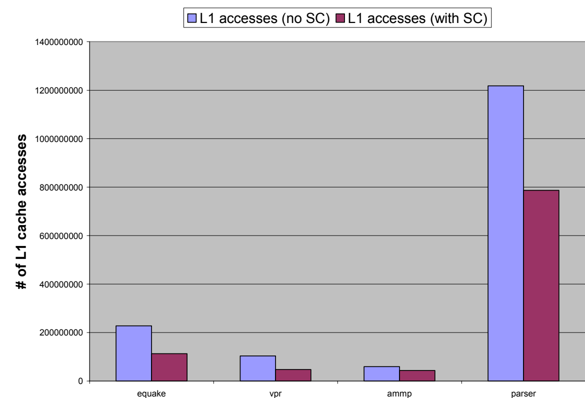

Figure 5: Total L1 accesses

Anywhere from 1/3 to 2/3 of the memory accesses are stack data that the stack cache would handle. When the stack cache is enabled it greatly relieves cache conflict problems in the L1.

5.4 Varying L1 associativity

With the stack cache handling a great deal of data instead of the L1, we speculated that the L1 could potentially afford to have a lower associativity and therefore lower latency. The data in Figure 6 shows the effect of decreasing the L1 data cache associativity to 2-way, while also decreasing the latency from 3 cycles to 2 cycles. Lower results are better.

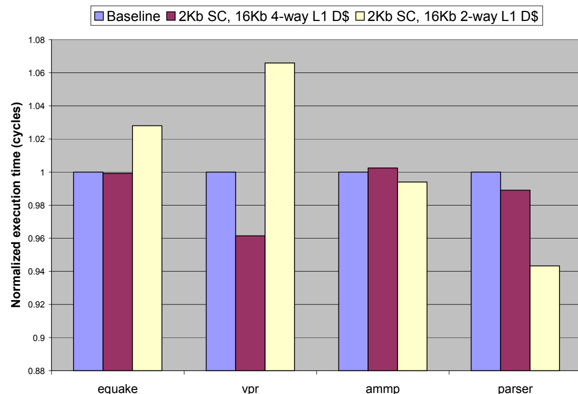

Figure 6: Varying L1 associativity

Clearly the results are mixed. The first two benchmarks show a marked decrease in performance with the lower associativity, indicating that there are still a great deal of conflict misses within the L1 even without stack data taking up space. The second two benchmarks indicate that the lower latency of the 2-way L1 offset the lower associativity. The data is inconclusive for determining if the L1 should be made faster and simpler when a stack cache is used.

5.5 Varying L1 latency

We are also interested in how the stack cache compares to the traditional L1 data cache if the stack cache did not have its lower latency as an advantage. The following test examines a 1 cycle, 16Kb L1 with and without a stack cache, compared to the baseline. The test was run only on the *equake* benchmark.

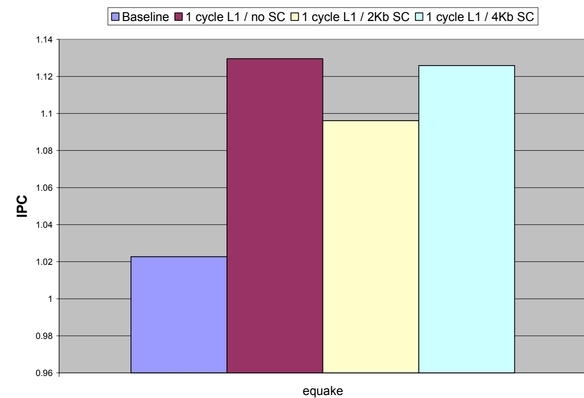

Figure 7: Varying L1 latency

The test clearly indicates that even the 4Kb stack cache doesn't compare to a fast 16Kb L1 data cache, and in fact hurts performance when used in conjunction. This is likely due to the prefill/spill mechanism, which rears itself again as an imperfection in our simulator code. The mechanism as we implemented it doesn't look ahead in the instruction stream for Stack Pointer modifications, so when the stack window has to move the stack cache locks all accesses until it writes all dirty, evicted data to the L2.

If the stack cache were doing its job right, the extra space available for data to be cached should increase performance, as well as the latency-hiding feature of the prefill/spill unit. This shortcoming in our simulation does not indicate a weakness of the stack cache, but rather the strength of a good prefill/spill unit.

5.6 Dividing hardware resources

The ultimate demonstration of the stack cache's effectiveness is if it outperforms the baseline given the same overall architectural resources as the baseline. This means that the stack cache will have to take precious cache space away from the L1, which seems to indicate a deleterious affect on performance in general.

However, combined with the stack cache, a reduced L1 could still be quite effective. The following data shows the effect of splitting a 4Kb L1 data cache into room for a combined 2Kb stack cache and 2Kb L1 data cache.

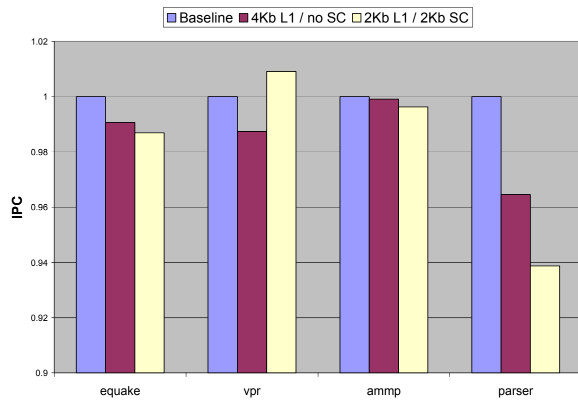

Figure 8: Shared resources

Clearly a 4Kb L1 with no stack cache should perform worse than a 16Kb L1 with no stack cache (the baseline configuration). However when the 4Kb L1 is split into a smaller L1 and a 2Kb stack cache it performs worse in most cases. This is in part due to the fact that the L1 is under so much strain already at 4Kb, that reducing it to 2Kb only helps when the stack cache handles more than half of the L1 references (as is the case in *vpr*, see Figure 5). However, the performance decreases are very small here.

Another factor in performance degradation is the fact that the large difference in a 2Kb and 4Kb stack cache was unknown before we ran this benchmark, so we were unable to measure the performance of a split 4Kb L1 and 4Kb stack cache. Additionally, the simulator prevented us from defining an L1 cache that was not a power of two. Our preferred test would have taken the 16Kb L1 baseline and split it into a 12Kb L1 and 4Kb stack cache.

6. Concluding remarks

Our first hypothesis is an architectural factor that we can't directly simulate. However we demonstrated that a stack cache can be indexed as a contiguous buffer of memory, which means it can be implemented as a direct-mapped cache. This, and its small size, allows us to trust that a stack cache could be implemented with 1 cycle access times.

Our second hypothesis focuses on how much stack data was accessed compared to heap data. Our benchmark data shows that anywhere from 1/3 to 2/3 of the accesses to the L1 cache are stack data, which can be removed from the L1 and directed to the stack cache. The L1 data cache likely benefits greatly from having decreased cache conflicts from stack data.

Our third hypothesis was verified indirectly. We discovered how much an effective prefill/spill unit could increase performance when we implemented a poor one. The L1 latency benchmarks, indicated by Figure 7, show that a poor prefill/spill unit makes the stack cache perform much worse than expected – however it is still on par with the performance of a fast L1 data cache. This indicates that an effective spill/fill unit could make the stack cache outperform even the fastest L1 data cache.

6.1 Is it worth it?

The stack cache appears to be an effective means of accomplishing our hypotheses. The question is, are they important objectives? The most important factor in modern architectures is still raw performance, and the results in Figure 3 indicate that a 4Kb stack cache (or 2Kb stack cache with effective prefill/spill) provides a noticeable performance improvement across the board. With more effective prefilling/spilling the stack cache could perform even better.

Additionally as the L1 data cache becomes slower with the uneven scaling of transistors versus transmission lines, it will become increasingly important to use more small, independent structures for fast data caching. The stack cache is effective at a much smaller size than the L1 data cache, and it has a significantly higher hit rate. This makes it better suited for scaling

All evidence indicates that a stack cache is an extremely powerful structure. It could be especially effective in architectures that are more reliant on stack accesses, such as IA-32 (x86), due to more register pressure or stack-based calling conventions.

References

[1] - Keith D. Cooper and Timothy J. Harvey, "Compiler-Controlled Memory", *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.