Spills, Fills, and Kills

An Architecture for Reducing Register-Memory Traffic

Mattan Erez
mattan.erez@stanford.edu

Brian P. Towles
btowles@stanford.edu

William J. Dally
billd@csl.stanford.edu

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

In a typical integer application, 40% of all memory references are generated by the compiler to: save and restore register values at procedure boundaries, pass parameters, and handle register allocation failures. We remove these compiler memory references by augmenting a conventional architecture with a spill name space and separate spill, fill, *and* kill *instructions to access this space. These instructions facilitate compiler name generation while allowing the hardware to take advantage of two key properties of compiler references. First, there is great locality in compiler references and a 128-entry* spill store *is able to capture over 99% of all spills and fills. Second, spills and fills use a simple addressing mode that permits addresses to be resolved in the decode stage of the pipeline, increasing ILP. The combination of a reduction in memory traffic and an ILP increase results in speedups of up to 40%.*

1. Introduction

Modern processors strive to execute an increasing number of instructions per cycle. This stresses an already overburdened memory system and requires more ILP to be extracted from the code. Both of these problems can be addressed by specializing an architecture to deal with two distinct categories of memory references: *user references*, which are the set of memory references explicitly programmed in a high-level language; *compiler-generated references*, which are the product of the compilation process and are inaccessible to the user. We refer to this set of compiler references as *spills* and *fills*, for memory writes and reads respectively.

The source of spill and fill operations is inadequate support for compiler-generated variable names. Architectures traditionally provide the compiler with a register name space and a memory name space. If the register space is too small or if names cannot be expressed in terms of static registers, the compiler must use memory names to represent these values. These memory names are accessed using store and load instructions resulting in spills and fills. Allowing the compiler to clearly delineate between its spills and fills and user stores and loads, enables the elimination of nearly all compiler-generated references from the memory system, which account for almost half of a program's total memory references.

We introduce a spill name space to facilitate efficient compiler naming and three new instructions that access this space. *Spill* instructions bind a value with a name in the spill space, *fill* instructions access values bound to particular spill names, and *kill* instructions discard values from the spill name space. Spilled values are kept in a small *spill store*, allowing fast hardware access. Kills indicate the last-use of a compiler name, which frees storage space in the spill store and prevents spilled values from being exposed to the memory system. The high locality of spill-fill pairs combined with the last-use information allows a 128-entry spill store to filter 99% of all spilled values from the memory system.

Spill, fill, and kill instructions use simplified indexing compared to general load and store instructions. All three instructions form variable names by specifying a constant displacement from a *spill-fill pointer* register. This enables the resolution of names early in the pipeline, allowing fills to be satisfied before they reach the execution stage, thus increasing effective ILP.

Taking maximal advantage of the spill, fill, and kill instructions reduces memory traffic, increases ILP, and provides a natural partition for several microarchitectural structures. The resulting *spill-fill-kill architecture* (SFKA) eliminates up to 45% of memory references with a small, 64-entry *spill store*, while providing a speedup of as much as 40%.

The remainder of the paper is organized as follows. In Section 2 we give a definition of spills and analyze their distinct code properties. Section 3 presents the proposed architecture, compiler modifications, and microarchitecture. We describe our evaluation methodology in Section 4 followed by experimental results and discussion in Section 5. Comparisons to related work appear in Section 6, followed by our concluding remarks and future research directions in Section 7.

2. Spill Definition and Related Code Properties

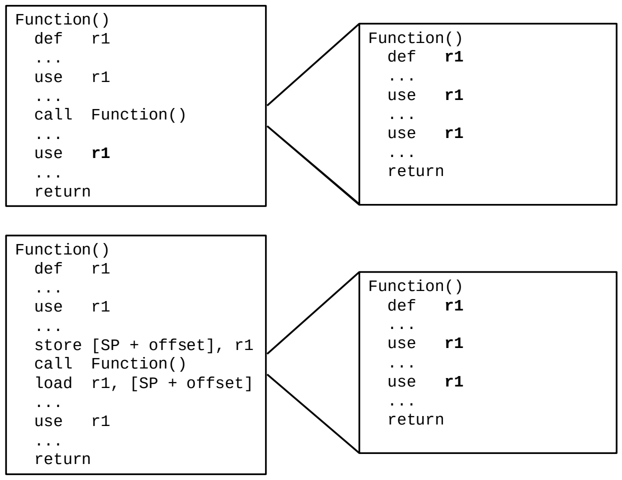
2.1. Spill Definition

Compiler-generated memory references are necessary for a set of three common occurrences in procedural languages:

- *Register allocation failures* occur when there are too few architectural registers to hold the set of live variables, and variables must be swapped between registers and memory.
- *Register saves and restores* occur across function calls, allowing the body of a called function to access the entire set of registers and decoupling the register allocation of different functions.
- *Parameter passing* allows function arguments to be stored to a memory location if they cannot be passed in registers.

Spills cannot be eliminated by simply increasing the number of architectural registers. While additional registers expand the static namespace, some variables require *dynamic names* – names dependent on run-time information. Since architectural registers are statically named, the compiler conventionally relies on memory for dynamic name generation, resulting in spills and fills. As an example, consider the necessity of dynamic naming for a recursive call (Figure 1). The value in register $r1 is live when the recursive call is made. The child function uses $r1 again, overwriting its previous value. To avoid this, a store instruction is introduced that saves the value into a unique memory location derived from the current context (e.g.. the stack pointer). Other cases that require dynamic naming include functions that may be called from different code fragments, such as library functions, and functions that accept a variable number of arguments.
Although compiler spills and fills are traditionally translated to store and load instructions, they are fundamentally different. Spills and fills are used as a *private naming mechanism*, disjoint from all other memory accesses. However, general memory operations are subject to architectural restrictions, including ordering and coherency, required for inter-thread and inter-process communication. Also, while spills and fills require dynamic names, their addressing needs are more limited than general load and store instructions. All compiler-generated references can be handled by specifying an immediate displacement from a *spill-fill pointer* register, similar in concept to a stack pointer.

There are two cases where the compiler cannot guarantee disjointness, which are therefore excluded from the set of spills and fills: variables whose names are available to the user, through an operation such as the C "address-of" (&) operator; all arrays, which require indexing that is more general than the register plus constant offset used to access elements on the stack and is less amiable to hardware acceleration (Subsection 3.3.1).



```
Function()
  def   r1
  ...
  use   r1
  ...
  call  Function()
  ...
  use   r1
  ...
  return
```

```
Function()
  def   r1
  ...
  use   r1
  ...
  use   r1
  ...
  return
```

```
Function()
  def   r1
  ...
  use   r1
  ...
  store [SP + offset], r1
  call  Function()
  load  r1, [SP + offset]
  ...
  use   r1
  ...
  return
```

```
Function()
  def   r1
  ...
  use   r1
  ...
  use   r1
  ...
  return
```

Figure 1: Example of dynamic naming

2.2. Spill Related Code Properties

Spills and fills can comprise a substantial fraction of a program's instructions. Figure 2 shows a categorization of dynamic loads and stores into regular memory operations, stack based memory operations (non-spills), and the three spill / fill types. The data was gathered from the 8 integer applications of the SPEC95 benchmark suite. The programs were compiled with a modified version of GCC [GCC] for the MIPS-like architecture of Simplescalar [Burg97], which uses 32 general-purpose, integer registers. The height of the bars represents the fraction of memory stores/loads out of the total number of executed instructions. Applications such as PERL and Vortex show over half of their memory writes are spills and one-third of their reads are fills. Vortex has an especially high percentage of parameter passing since it frequently uses large data structures as arguments to its functions. M88ksim, IJPEG, and Compress show lower percentages of spills and fills, from less than 1% to about 18%. Compress in particular contains no function calls or register allocation failures in its single inner loop, giving no opportunity for spills.

The types of spills and fills seen in these programs are partially dependent on the number of registers in the target architecture. The percentages shown in Figure 2 reflect the behavior for a particular thirty-two-register architecture: save/restore spills and fills are more frequent than register allocation failures. More available registers lead to fewer register allocation failures, but more register saves and restores are required for function calls. Opposite trends occur with fewer registers.

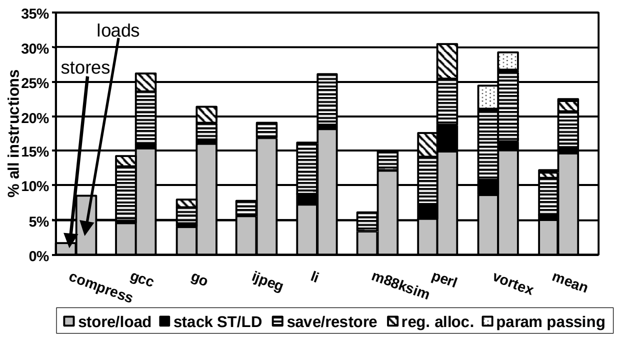


Figure 2: Memory operation classification

Spill-fill pairs have a high locality of reference, and over 99% of these references can be captured with 128 storage locations. This is supported by the cumulative histogram of the number of storage locations required to satisfy a spill-fill pair (Figure 3). The vertical axis shows the percentage of all spills for which the final matching fill could be found within a structure containing a certain number of storage locations varied along the horizontal axis. The number of storage locations is calculated by counting the number of *live spills* between every spill-fill pair, where a spill is considered live until it is read by its final matching fill. The figure shows data for several SPEC95 applications as well as the average for all the integer applications, weighted by the number of spills and fills in each application ('mean' line).

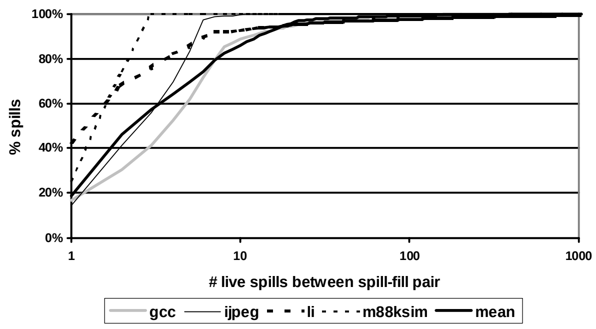


Figure 3: Dynamic spill distance

3. Architecture

A simple addition to the processor ISA can enable the compiler to communicate its intent for internally generated references, and allow the hardware to take advantage of the distinct properties of spills and fills. We present ISA extensions in Subsection 3.1, describe the necessary compiler modifications in Subsection 3.2, detail the spill-fill-kill microarchitecture in Subsection 3.3, and provide an alternate implementation in Subsection 3.4.

3.1. ISA Extensions

A set of extensions is provided for a standard RISC ISA tailored to the needs of compiler name generation and independent from the underlying implementation details. The naming interface is organized like a traditional stack[1], which allows for minimal impact on compiler design. The extensions specify a *spill name-space*, disjoint from the user memory namespace, and provide instructions to access it based on an immediate displacement from an architectural *spill-fill pointer* register (SFP), analogous to the stack pointer. Unlike a normal register, write access to the SFP is restricted and only specific instructions are allowed to modify it:

`setsfp` *`imm/reg`*	Set SFP to an immediate or register value.
`incsfp` *`imm`*	Increment/decrement the SFP.

The `setsfp` and `incsfp` instructions provide all the functionality the compiler requires to manage a stack. For example, the SFP can be initialized with `setsfp` at the beginning of each thread of execution and then modified by `incsfp` as the compiler creates and releases names. By restricting accesses to the SFP to these two instructions, name generation is simplified and more readily optimized in a hardware implementation (Subsection 3.3.1).

Additional instructions facilitate allocation and use of the compiler-generated names and their values:

`spill reg, [sfp+imm]`	Bind the value from *reg* with the specified name
`fill [sfp+imm], reg`	Read value bound to the specified name into *reg*.
`kill [sfp+imm]`	The value bound to the specified name can be discarded.
`fkill [sfp+imm], reg`	`fill` after which the value can be discarded.

The `spill` and `fill` instructions are different from store and load instructions. The only architectural guarantee for spills and fills is provided for accesses with the same name – a fill returns the value from the most recent spill, unless a `kill` or `fkill` was encountered after the spill. A `spill` does not write the value into a memory location, but only binds the value with the specified name. This allows the value associated with a

[1] The data structure used is not a stack in the strict sense – a small set of the top most elements, called a *frame*, can be accessed randomly.

name to be discarded using a kill instruction. Store instruction values, conversely, cannot be discarded because they are subject to memory consistency and coherency restrictions. Finally, spill names form a logically separate address space from user accesses, but if a spilled value must be written to memory, the name is used as a full virtual address. A matching fill then retrieves the value by using its own name as an address.

3.2. Compiler Modifications

Adapting a compiler to incorporate the ISA modifications is straightforward. First, compiler-generated memory accesses are marked in the stages they are created: failures during register allocation, and register save/restore and parameter passing during function call insertion. Except for compiler-generated variables whose names are available to the user or array variables (Subsection 2.1), all accesses created in these stages are noted as spills. These exception cases can be gathered during the compiler's parsing pass. Once all code optimizations have occurred, any remaining, marked compiler references are converted to spill and fill instructions.

After all spills and fills have been generated, live variable analysis is used to determine which fills can be converted into `fkill` instructions. The lifetime of a compiler-generated variable begins when it is first spilled and the variable is considered "dead" after being used by its last fill. The liveness of a particular variable is dependent on control flow, so conversion of a `fill` to `fkill` can also be control dependent. For example, if a variable is live upon entry to a basic block and is dead along all exit paths of that block, the last fill of that variable in the block can be converted to a `fkill` (Figure 4a). However, it is also possible for a variable to be dead along some, but not all, of the exit paths (Figure 4b). In this case, an extra `kill` instruction should be inserted that post-dominates the fill. There are several potential locations for inserting these `kill` instructions. Two possibilities are either along all the paths where the variable is dead (Figure 4c) or in the return block, if the function has a single exit point (Figure 4d).

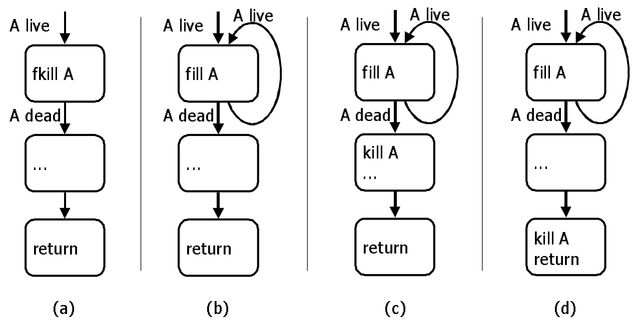


Figure 4: Insertion of kill instructions

Conveying dead variable information incurs almost no code size cost, since existing `fill` instructions can be converted to `fkill` instructions for 99% of spilled values. However, if code size is a primary concern, the `kill` instructions can simply be left out since they are not required for functional correctness. The resulting performance impact is minimal because 99% of spilled values are still eliminated with `fkill` instructions.

The SFP is initially set to the virtual address at the top of the program's stack with the `setsfp` instruction. Then each compiler-generated reference is allocated a stack entry relative to the SFP. Spilled values on the stack are accessed with the `spill` and `fill` instructions, while any remaining compiler references are also placed on the stack, but accessed with load and store instructions. In this way, a dense memory layout of spill and non-spill values is achieved, and the addition of explicit spill and fill instructions requires no additional virtual address space. Finally, each procedure can require additional stack space, so upon procedure calls and returns the SFP is incremented and decremented using the `incsfp` instruction.

3.3. Microarchitecture

There are many possible ways of implementing our proposed ISA extensions, including simply treating fills as loads and spills as stores. This section presents a spill-fill-kill microarchitecture that fully exploits the properties of fills, spills, and kills to increase performance by reducing memory traffic and increasing ILP.

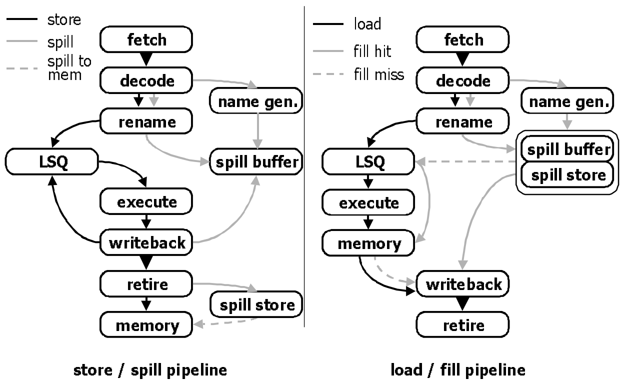


Figure 5: Architecture pipeline

Our microarchitecture is summarized in Figure 5 separated into a store/spill and load/fill pipeline. A regular store proceeds from decode, through the load store queue (LSQ) to address generation, is written back to the LSQ when both address and data are available, and finally is committed to memory when the store retires. A spill, on the other hand, may forgo address generation in the execution units since the restricted use of the SFP allows the name generation to occur in the front-end (Subsection 3.3.1). It is then allocated an entry in the *speculative spill* *buffer* (3.3.2) instead of in the LSQ, and finally, when the spill retires, its value is written to the *spill store* instead of to memory (3.3.3). A spilled value is exposed to the memory system only when a spill is evicted from the spill store. A regular load is processed much like a store, but must also undergo disambiguation before reading from memory. A fill behaves quite differently. Like a spill, the fill's name is obtained before execution and is used to query the spill store and speculative spill buffer in parallel. On a hit, the fill skips execution and goes directly to the writeback stage with the value obtained from the spill store. If the name was not found, the fill proceeds down the pipeline as a regular load with two differences: 1) its address has already been calculated; 2) it need not undergo disambiguation since its address is guaranteed not to collide with any store (spills are not written to memory as stores).

3.3.1. Hardware Name Generation

For the hardware to take full advantage of the compiler information, it must discern spill names in an early pipeline stage. This is achieved by employing a simplified version of the *register tracking* technique presented in [Beke00]. Register tracking allows calculation of spill names relative to the SFP as early as decode time with the addition of a simple adder to the decode stage (or any other front-end stage).

The `setsfp` instruction sets the SFP with either an immediate or register value. Once the SFP has been set, updates to it are performed with `incsfp` instructions that use an immediate operand. These `incsfp` instructions can be "pre-executed" in the decode stage by adding the immediate to the current SFP. Therefore, subsequent references to the SFP can be resolved in the decode stage as well. Using a register operand for `setsfp` requires information to be passed from the execution stage to the front-end, but since this is only required for process initialization the processor can simply stall until this value is ready without a noticeable performance impact.

3.3.2. Speculative Spill Buffer

The use of a spill store in conjunction with speculative execution requires a *speculative spill buffer* (SSB) to store speculatively processed spill instructions. Regular loads and stores have similar requirements and the *load store queue* (LSQ) already exists to deal with them. The SSB is a simplified *store buffer* portion of the LSQ, which tracks the order of spills and records their names and values. Each entry of the SSB has a *name field*, *value field*, and a *value bit* indicating whether the spilled value is ready. Like a store buffer, the SSB is a prioritized CAM that returns the most recent entry that matches a given name.

After a spill's name is resolved through hardware name generation, a SSB entry is allocated and tagged with the name. Spills are not put in the LSQ's store buffer, thus reducing the pressure on this expensive resource. Also, since spills are disjoint from loads, no disambiguation between the SSB and loads in the LSQ needs to occur, which provides an ILP increase by removing these false dependencies. Spills then proceed to the rename stage to determine their source data register. If that value is ready, it is read from the register file, placed in the SSB, and the entry's *value bit* is set. Otherwise the name of the register that will produce the value is recorded in the SSB and the *value bit* is reset, indicating that this entry must be updated when the value is produced.

A fill receives its name from the name generator and queries the SSB and spill store in parallel. If a matching entry is found in the spill store only, the value is read and written back to the fill's destination register. This process occurs in the front-end, thus having the same effect as a zero-cycle operation for dependent instructions and ILP is increased.

There are two possibilities for when a SSB entry matches the fill. If the value bit is set, the spilled value is immediately written into the fill's destination. Otherwise the fill is issued as a register-to-register move instruction, from the spill's source register (stored in the SSB entry) to the fill's destination register. Dependent instructions can use this value once the register move executes.

If the fill misses in both the SSB and spill store, it is treated in a manner similar to a regular load. However, the fill's virtual address, which is simply its name, has been calculated in the front-end, and is already disambiguated. Disambiguation is guaranteed since the fill and store address spaces are kept disjoint by the compiler, and any outstanding spill to the same address would have resulted in a SSB hit.

The disjoint address spaces for spills and stores also allows a natural partitioning of the traditional store buffer into the store buffer and speculative spill buffer. This may become important as these structures are prioritized CAMs, which must grow significantly larger with each generation to support the increased level of instruction and memory level parallelism, and may become a cycle-time limiter if not partitioned.

3.3.3. Spill Store

The *spill store* is used to filter spills before they are issued to memory, and to provide efficient access to recently spilled values. It is arranged as a fully-associative or set-associative structure for holding spilled values and is tagged by the spill names. Once a spill retires, its speculative name and value are transferred from the SSB and committed to the spill store without affecting the memory system. Later fills read this value and make it available to dependent instructions with low latency. Most spills are kept from ever reaching the memory system by taking advantage of the `kill` and `fkill` instructions. Whenever a `kill` or `fkill` instruction is retired, the spill store entry matching it is invalidated. In general, there may be more live spill values than can be held by the spill store. In this case, the least recently used entry is evicted and written to actual memory for later retrieval by a fill. The spill store uses a separate datapath to the memory subsystem and the spill's name as a virtual memory address. In order to prevent the spill store from stalling retirement, when a TLB or cache port is not available, a high watermark can be used to govern early eviction from the spill store utilizing free memory slots.

The spill store does add extra state to the processor, which potentially increases the overhead of a context switch. This only applies to context switches that affect the virtual address mapping (i.e. a process switch) because the compiler generates unique spill names for each thread within a single address space. When the virtual address space is changed, the data in the spill store must be written to memory to prevent aliasing within the new address space. An extension to the spill store could eliminate this problem by adding a process ID field to each entry. Thus, a name match must include the process ID, but the structure is not flushed on a process switch.

3.4. Alternative Microarchitectures

The microarchitecture presented in Subsection 3.3 takes full advantage of the compiler information by adding storage locations for the spilled values. An alternate architecture is based on the observation that all spilled values reside in some physical register at spill time. We propose to use these physical registers as a logical spill store and speculative spill buffer.

The *spill alias table* (SAT), much like the *register alias table* used for register renaming [Dief99], maps spill names to the physical registers containing the spilled values. When a spill instruction is encountered the source data register of the spill is associated with the spill name in the SAT. A later fill performs an associative lookup for its name in the SAT and uses the returned physical register. If a fill occurs from a live physical register (a register that may still be referenced directly by future instructions) the fills destination can be remapped to it using the *physical register reuse* method of [Jour98]. Physical register reuse allows reallocating a physical register as the destination of a later instruction, provided it is guaranteed to produce the same value. If the physical register is no longer live and has not been re-allocated as a new destination register, it can be used immediately as the destination register of the fill, and the source of any dependent instructions. To delay the re-allocation of these spilled values, physical registers that do not hold spilled values should be allocated first (by keeping two free-lists for instance).

The speculative state of the SSB is now kept in a *speculative spill alias table* (SSAT), similar to the *speculative register alias table* (SRAT), and in the physical registers containing the spilled values. Like the SRAT, the SSAT is checkpointed on every branch, and can be recovered once a branch misprediction is detected. The complication of this scheme lies in actually writing spilled values to memory when a spill physical register must be re-allocated. The only way to write to memory is by injecting a store instruction that must use the register as its source, thus stalling the allocation stage.

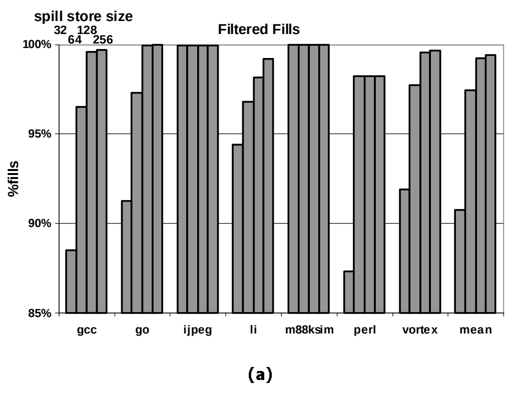
4. Methodology

The compiler modifications discussed in Subsection 3.2 were incorporated into GCC 2.6.3 [GCC]. Changes to GCC included marking of spill/fill memory references, live variable analysis for insertion of kill instructions, and code generation for the new instructions. In the cases where `fkill` instructions could not be used to indicate the last use of a spilled value, the compiler inserted `kill` instructions at the function exit point. GCC was targeted for the Simplescalar PISA, a MIPS-like architecture with 32 general-purpose registers. The resulting binaries were simulated using the execution driven Simplescalar Toolset [Burg97].

We used the eight integer applications of the SPEC95 benchmark suite. These applications do not particularly stress the register allocation failure spills, but do have many function calls with their associated spills and fills. Each program was executed for 100 million instructions of warm-up after which 100 million additional instructions were used to produce the results.

First, relevant code properties that were mostly independent of the microarchitectural details were analyzed. The Simplescalar functional simulator was used to collect these code properties and ascertain ILP. To determine ILP, an idealized, non-pipelined, execution engine was simulated. All dependencies were honored, but all instructions, including memory operations, executed in a single cycle. ILP was measured as the number of instructions that were issued every cycle from a fixed-size instruction window. The fetch unit was also idealized, refilling the instruction window at the start of every cycle. Instructions directly dependent on a fill executed on the same cycle as the fill.

Finally, to illustrate the spill-fill-kill architecture in a more concrete context, the Simplescalar out-of-order simulator was modified to handle spill and fill instructions and to include a speculative spill buffer and spill store. The baseline architecture was a 4-wide, 64-deep instruction-window machine, with a 15-bit gshare branch predictor, and five pipeline stages. A relatively large four-way, 64KB, first level data cache backed by a 1MB unified second level cache was used to minimize any second-order interference effects of stack accesses, which could have been advantageous when spills were eliminated from the memory system. No additional space was provided for the SSB and the total size of the LSQ and SSB remained constant. A 64 entry, fully-associative, spill store was used, providing a good tradeoff of extra state and coverage. Also, since we found that additional



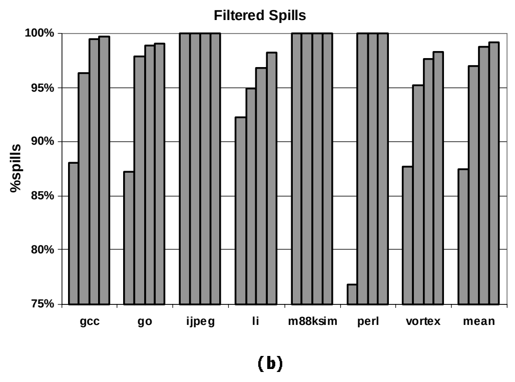


Figure 6: Fraction of fills and spills satisfied by the spill store

`kill` instructions provided little benefit over `fkill` instructions they were not inserted into the code used for performance evaluation.

5. Results

5.1. Code Properties

The classification of memory reference instructions into spill/fill types was presented in Subsection 2.2, as well as the number of live spills metric (figures 2 and 3). We now proceed with code properties that apply specifically to the spill-fill-kill architecture (SFKA). Since Compress has no spills it is not depicted in any of the figures in this section. However, the data gathered for Compress is taken into account when calculating the mean values.

Figure 6a shows the *fill hit rate* – the fraction of all fills that are captured by the spill store and do not go to the memory subsystem. The horizontal axis is divided into groups of growing spill store size from 32 to 256 entries, and the vertical axis is the fill hit rate. The hit rate is quite high, approximately 91%, for a small store of only 32 locations, and saturates at over 99% with 128 storage locations. Figure 6b shows the fraction of all spills that were successfully eliminated by a `fkill` or `kill` instruction. That is, the final use of the spilled value occurred while the value was still in the spill store, and it was successfully filtered from memory. Again, with only 32 storage locations an 88% spill reduction rate is observed, and increasing the storage to 128 captures over 99% of spills. Note that over 99% of the values killed were by a `fkill` instruction, where `kill` instructions increased the code size by at most 0.5%, and less than 1% to the number of instructions fetched. Therefore, not inserting `kill` instructions is the better option for this configuration and applications.

Figure 7 depicts the same results in a different manner, as the number of memory operations that were removed from the memory subsystem. Again, the horizontal axis shows growing spill store size, and the vertical axis shows the fraction of memory operations removed. Each bar has four parts, the bottom two are for memory writes (writes that went to memory and the ones filtered out), and the top two are for memory reads. With 32 entries, 35% of all memory operations were satisfied by the spill store (45% of writes and 28% of reads), and little benefit is seen beyond 128 entries that capture 38.5% of all memory operations (53% of writes and 31% of reads).

Figure 8 presents the ILP increase due to partial memory disambiguation and early resolution of fills. The vertical axis is the harmonic mean of the available ILP in the benchmarks, measured using the idealized execution engine. Along the horizontal axis the scheduling window size, memory disambiguation mechanism, and use of SFKA are varied. Note the very high ILP rates possible due to idealized execution, perfect branch prediction, and instruction fetch. Without performing perfect memory disambiguation (in perfect disambiguation loads only wait for prior stores to the same address), the SFKA increased ILP by 13-17% as the window size was increased. With perfect memory disambiguation higher ILP was available and using explicit spill, fill, and kill instructions allowed up to a 31% increase in mean ILP and over 50% in some applications (GCC and Vortex). An interesting observation is that the SFKA provided little disambiguation. Without full memory disambiguation many loads could not proceed and impeded ILP extraction.

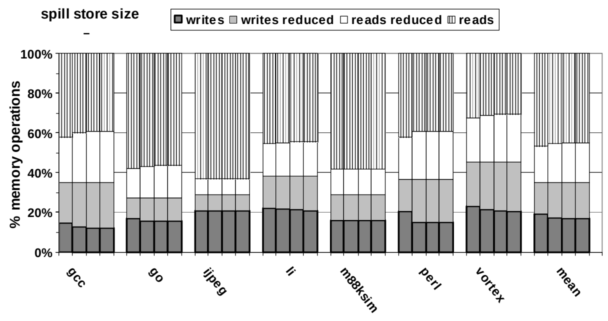


Figure 7: Memory operations serviced by the spill store

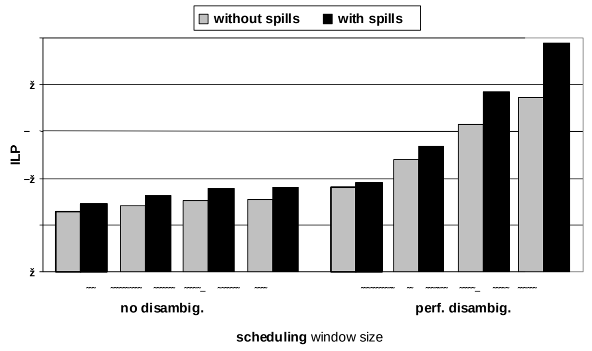


Figure 8: Harmonic mean of ILP increases

5.2. Speedup Analysis

We chose several configurations to show that the SFKA allows significant speedups for the microarchitecture simulated by Simplescalar. These experiments demonstrated that the measured code property benefits of Subsection 2.2 lead to speedups in execution, but were not meant as a study of the various implementation tradeoffs involved.

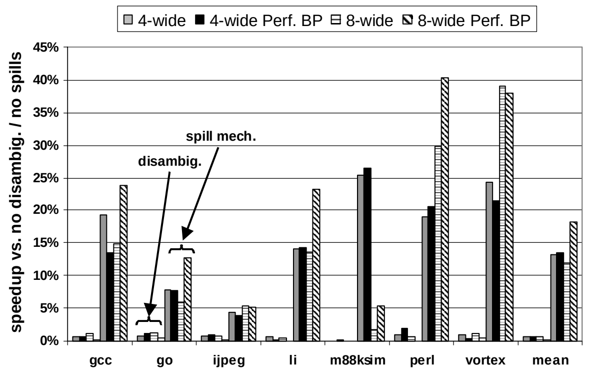


Figure 9: Speedup vs. machine width

Figure 9 shows the speedup from using the SFKA, for varying machine issue widths and scheduling window sizes. Each group of eight bars is sub-divided into two groups: the left four bars are for speedup due to perfect memory disambiguation, and the right bars are for speedup gained with SFKA. Each four-bar group represents growing machine width with the left most bar corresponding to the 4-wide, 64-deep baseline configuration, the second bar to the same configuration with perfect branch prediction, and the two right bars to an 8-wide, 128-deep machine with gshare and perfect branch prediction respectively. We simulated perfect branch prediction to shift the bottlenecks towards the memory system, thus better exposing the advantages of using SFKA. The speedup was calculated for each of the four machine configurations independently, where the baseline for each configuration was the same machine without both memory disambiguation and SFKA. The memory access reduction and ILP increase observed above lead to substantial speedups ranging from 5 to 40%, whereas the established technique of memory disambiguation improved performance by less than 1% on all configurations simulated. The greater speedups achieved on the wider machine indicate that the extra exposed ILP plays an important part in the speedups achieved, since these configurations provide more performance headroom when ILP can be extracted from the application. Because the graphs display speedup on different baseline configurations, the performance advantage of the SFKA can decrease as the baseline performance improves.

In order to isolate the effects of effective memory bandwidth increase from ILP increase, the number of available L1 cache ports was varied (Figure 10). Again, the speedup was calculated for each port configuration by measuring performance with and without the SFKA. As expected, the speedup decreased for configurations with more available memory ports, since the pressure on this resource was decreased regardless of the SFKA. However, the speedup remained considerable (15% on average) indicating that the extra ILP was beneficial. M88ksim, for example, benefited most from the increased memory bandwidth (shown by the sharp decline in speedup as memory ports are added). Whereas Vortex's performance was improved mainly by increasing ILP, since even with as many memory ports as issue slots a 20% speedup was observed. Another advantage of the small spill store is that its access time is lower than that of a typical first level cache, but all the experiments shown here assume a single cycle latency for the first level cache.

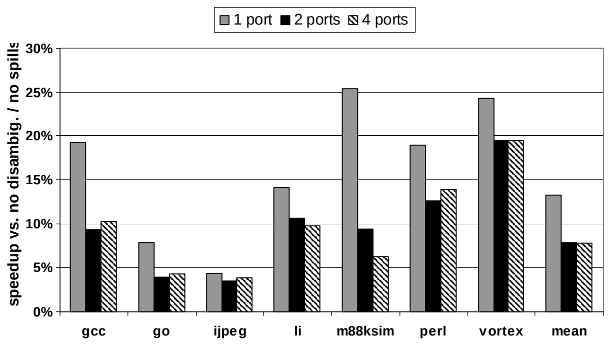


Figure 10: Speedup vs. memory bandwidth

6. Related Work

In this section we will briefly describe other work related to register spilling. Subsection 6.1 is dedicated to hardware–software interaction approaches that deal primarily with facilitating *save/restore* and *parameter passing* spills and fills. Ideas focused mostly on *register allocation failure* spills are addressed in Subsection 6.2. Finally, hardware only mechanisms designed to alleviate the spill problem are described in Subsection 6.3.

6.1. Save/Restore Handling

The original work on RISC recognized the importance of providing architectural support for high-level languages and particularly for procedure call/return. Patterson and Sequin suggested *overlapped register windows* to facilitate call save/restore and function [Patt81], an idea similar to the multiple register banks in the BBN C/70 [Kral80] and implemented in the SPARC architecture [Weav94]. In essence, each time a call is made, a fresh set of registers is presented to the new context, inherently saving the previous set without explicit memory operations. “Spills” of register windows occur when all windows have been used, and since the liveness of particular elements is unknown, the entire register window must be stored to memory. Two key disadvantages of register windows are: register frames are of fixed size, regardless of the actual usage in the called function; the physical register file size grows with the number of frames supported, limiting scalability.

Variable size register frames, are similar to register windows, but address the drawback associated with their fixed size. On a function call, an *allocate* instruction is used to create a register frame in a large physical register file, which provides room for several frames. When registers wrap around to the beginning of the register file, which is organized as a circular buffer, the old values are spilled to memory by the hardware. This mechanism was implemented in the Am29000 [AMD87], and more recently in the Intel IA64 architecture [Inte00]. In IA64, the *register stack engine* works in the background saving and restoring registers as needed. A drawback of these techniques is that working set of architectural registers is expressed using an offset from a *current window pointer*. The problem of scalability is not directly addressed due the large size of the physical register file.

Shifting register windows is a variant of variable size windows, which removes the need for calculating a register's name from the current window pointer, and provides a mechanism for secondary storage to deal with scalability [Russ93]. Spills and fills to memory are handled by the hardware and occur as single memory accesses, not as entire frames. This requires the hardware to provide memory for spilled values, but the work does not discuss how this memory is allocated, or how a swapped-out context can later reload its values. The SFKA addresses all the drawbacks above while providing the same functionality, as well as handling register allocation and parameter passing spills, at the expense of adding explicit save/restore instructions.

Several other enhancements to register windows such as *Threaded Windows* ([Quam88]) and *Multi Windows* ([Scho95]) have been suggested in the context of multi-threaded architectures for dynamic management of many live contexts.

Huguet and Lang describe a mechanism to reduce save/restore traffic for a non-windowed register file [Hugu91]. The compiler performs inter-procedural register allocation to minimize register name overlap between caller and callee procedures. In addition, the hardware tracks which registers are in use by the caller to reduce the number of restores required after a call return.

[Mart97] proposed using dead value information, passed from the compiler using a *kill* instruction (a kill here is for an architectural register and not a dynamic compiler name), to reduce save / restore traffic to memory. Their idea is similar to [Hugu91] in that the callee only saves registers that are needed by the caller (still live in the caller context). Eliminating restores is more complex and involves additional hardware structures.

The *Named-State Register File* (NSF), which was introduced to facilitate fine grain multi-threading, is organized as a content addressable memory, where each register is named relative to its context [Nuth95]. As a result, there are no fixed frames and every context uses as many registers as needed, requiring additional instructions to manage the context-to-memory mapping. Each register is uniquely named, so the hardware can dynamically spill and fill values on demand, allowing the NSF to handle all spill types.

6.2. Register Allocation Failure Handling

The straightforward solution to the *register allocation failure* problem is to add more architectural registers, but this leads to implementation problems due to the increased instruction word size. In *register connection* [Kiyo93], instructions are introduced to dynamically re-map register identifiers (e.g. $r13). The compiler has access to more architectural registers at the expense of additional instructions to manipulate the register re-mapping. Alternatively, Lozano and Gao observed that the live ranges of many variables are entirely contained within the instruction window [Loza95]. They suggest that only variables with long live ranges get assigned architectural register names, and short-lived variables get instruction window based names. Both techniques increase the number of available static names, and do not address the problem of dynamic naming.

The concept of *compiler controlled memory* (CCM) was described in [Coop98], where the hardware provides a predefined memory structure for compiler use only. The idea is similar to the 16k compiler controlled SRAM of the Cray 2 [Simm88]. The compiler modification suggested is to use this memory as a store for spilled values. The paper does not discuss how a CCM can be used with dynamic naming but has features in common with our approach, such as a disjoint name space for compiler and user memory references.

6.3. Hardware Only Schemes

Although not strictly hardware only, the *stack cache* suggested in [Ditz82] dynamically allocates registers to values at the top of a hardware stack. More recently the idea was adapted to the picoJava processor [Ocon97]. Since these stack-based approaches lack architectural registers, explicit register allocation failure and register saves and restores are avoided. To prevent parameter passing spills, the stack frames of the caller and callee are overlapped.

In [Mosh97] and [Tyso97] mechanisms are presented that dynamically and speculatively detect store-load pairs. This information is then used both for memory disambiguation and for bypassing the store-load chain, allowing values to be forwarded directly from the producer to the consumer. [Mosh97] also notes the high temporal locality of many store-load pairs, and suggests the *transient value cache*, which is a structure similar to our spill store, but requires speculative access. Although they do not consider this in their papers, many of the predictable communication paths are likely to be spill-fill pairs dealt with by our static approach.

Cho et al. observes that access to *local variables* (all automatic variables) are decoupled from regular memory references [Cho98]. Based on this observation they suggest partitioning the memory related microarchitecture between a traditional *load store queue* and a *local value queue*, both of which have separate first level caches. They perform simulation studies on the potential of this technique based on perfect hardware classification of loads and stores into regular and local value references. They also abstractly discuss a compiler-architecture interaction approach, but do not provide a detailed mechanism for communicating the required information, and cannot guarantee that the decoupled accesses are truly to disjoint address spaces.

[Beke00] deals with early address resolution of loads and stores to the stack. They suggest a way for the hardware to safely track the value of the *stack pointer* register in an early pipeline stage. The address is then used to disambiguate some of the loads and to issue these loads earlier, which provides more effective ILP. Complications arise due to the fact that the stack pointer can be modified in any way, unlike our technique which restricts updates to the SFP, and provisions are made for when it becomes *untrackable*. The approach is later extended to track all registers, and to provide a separate *stack cache* used in parallel to the first level cache. This stack cache must be snooped by regular memory accesses since it is not guaranteed to be disjoint from regular loads.

7. Conclusions and Future Work

This paper identified two distinct sources of memory accesses: compiler-generated memory references, referred to as *spills* and *fills*, and user references. Analysis of the SPEC95 integer applications showed that a large fraction of a program's memory operations are spills and fills, and only a small amount of storage is required to hold nearly all spilled values. Based on these code properties, the spill-fill-kill architecture was proposed to improve cooperation between the compiler and the hardware, and provide a unified mechanism to deal with spills and fills through a set of ISA extensions. The new *spill*, *fill*, and, *kill* instructions were tailored for compiler-generated references, while preserving an abstract view of the underlying hardware details.

We modified GCC to take advantage of the ISA extensions and provided hardware constructs to exploit the newly provided information. The *spill store* was introduced as a specialized hardware structure that filters spill and fill references from memory. A small spill store of only 128 entries removed up to 45% of the memory references, while satisfying over 99% of the spills and fills. The *speculative spill buffer* extended the functionality of the spill store to support speculative execution, and also created a natural partitioning of microarchitectural structures such as the load-store queue. Finally, *hardware name generation* was integrated into the processor's front-end, which allowed fills to be serviced early in the pipeline, making their results available immediately to dependent instructions. Consequently, the dynamically extracted ILP increased by a factor of two or more. Hardware name generation could increase the effective execution bandwidth by saving load address calculations, and provided a measure of memory disambiguation. The improved code properties also translated into significant execution benefits, where some programs exhibited a speedup of over 40%.

The introduction of an intermediate spill name space between the static register name space and the completely general memory name space has uses beyond the elimination of compiler references. For example, a separate name space can be used as the target of high-latency memory loads. This allows a larger number of outstanding references than possible using a register name space. In a multiprocessor, a separate name space can hold data that is private to a given process and thus not subject to the consistency and coherency policies that must be applied to potentially shared data. Shielding private data in this manner reduces the load on the consistency mechanisms and eliminates a major source of false sharing. Developing techniques for compilers to identify and group the characteristics of these and other memory accesses, along with corresponding hardware mechanisms to capitalize on this information, is a source of future research.

References

[AMD87] "Am29000 User's Manual," *Advanced Micro Devices*, 1987.

[Beke00] M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen, "Early Load Address Resolution via Register Tracking," ISCA 27, June 2000.

[Burg97] D.C. Burger and T. M. Austin, "The Simplescalar toolset, version 2.0", Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[Cho98] S. Cho, P-C Yew, and G. Lee, "Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor," *ISCA 26*, June 1998.

[Coop98] K. Cooper and T. Harvey, "Compiler-Controlled Memory," *ASPLOS VIII*, October 1998.

[Dief99] K Diefendorff, "PC Processor Microarchitecture", Microprocessor Report, Vol. 13 No. 9, July 12, 1999.

[Ditz82] D. Ditzel and R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *ASPLOS I*, 1982.

[GCC] The GNU Project, "GCC Online Documentation", http://www.gnu.org/software/gcc/onlinedocs/ .

[Hugu91] M. Huguet and T. Lang, "Architectural Support for Reduced Register Saving / Restoring in Single-Window Register Files," *ACM Transactions on Computer Systems, Vol 9, No 1*, February 1991.

[Inte00] "Intel IA-64 Architecture - Software Developer's Manual," *Intel Corp.*, July 2000.

[Jour98] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification," *MICRO 31*, December 1998.

[Kiyo93] T. Kiyohara, S. Mahlke, W. Chen, R. Bringmann, R. Hank, S. Anik, and W-M Hwu, "Register Connection: A New Approach to Adding Registers into Instruction Set Architectures", *ISCA 20*, 1993.

[Kral80] M. Kraley, R. Rettberg, P. Herman, R. Bressler, and A. Lake, "Design of a User-microprogrammable Building Block," *13th Annual Workshop on Microprogramming*, December 1980.

[Loza95] L. Lozano C. and Guang Gao, "Exploiting Short-Lived Variables in Superscalar Processors," *MICRO 28*, Novemeber 1995.

[Mart94] M. Martin, A. Roth, and C. Fischer, "Exploiting Dead Value Information," *MICRO 30*, December 1997.

[Mosh97] A. Moshovos and G. Sohi, "Streamlining Inter-operation Memory Communication via Data Dependence Prediction," *MICRO 30*, December 1997.

[Nuth95] P. Nuth and W. Dally, "The Named-State Register File: Implementation and Performance," *HPCA 1*, January 1995.

[Ocon97] J. M. O'Connor and M. Tremblay, "picoJava-I: The Java Virtual Machine in Hardware," *IEEE Micro, Vol 17 No 2*, March-April 1997.

[Patt81] D. Patterson and C. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *ISCA 8*, 1981.

[Quam88] D. J. Quammen, D. R. Miller, and D. Tabak, "Register Window Management for Multitasking Applications," *21st Hawaii International Conference on System Sciences*, 1988.

[Russ93] G. Russel and P. Shaw, "Shifting Register Windows," *IEEE Micro, Vol. 13, No. 4*, August 1993.

[Scho95] T. Scholz and M. Schafers, "An Improved Dynamic Register Array Concept for High-Performance RISC Processors", *28th Hawaii International Conference on System Sciences*, 1995.

[Simm88] M. L. Simmons and H. J. Wasserman, "Performance Comparison of the Cray-2 and Cray X-MP/416," *Supercomputing '88*, 1988.

[Tyso97] G. Tyson and T. Austin, "Improving the Accuracy and Performance of Memory Communication through Renaming," *MICRO 30*, December 1997.

[Weav94] D. Weaver and T. Germand, "The SPARC Architecture Manual," *SPARC International, PTR Prentice Hall*, 1994.