Stack Value File: Custom Microarchitecture for the Stack

Hsien-Hsin S. Lee Mikhail Smelyanskiy Chris J. Newburn[†] Gary S. Tyson

Advanced Computer Architecture Lab
University of Michigan
Ann Arbor, MI 48109
{*linear, msmelyan, tyson*}*@eecs.umich.edu*

[†]PMD Architecture
Intel Corporation
Portland, OR 97124
cnewburn@ichips.intel.com

Abstract

As processor performance increases, there is a corresponding increase in the demands on the memory system, including caches. Research papers have proposed partitioning the cache into instruction/data, temporal/non-temporal, and/or stack/non-stack regions. Each of these designs can improve performance by constructing two separate structures which can be probed in parallel while reducing contention. In this paper, we propose a new memory organization that partitions data references into stack and non-stack regions. Non-stack references are routed to a conventional cache. Stack references, on the other hand, are shown to have several characteristics that can be leveraged to improve performance using a less conventional storage organization. This paper enumerates those characteristics and proposes a new microarchitectural feature, the *stack value file* (SVF), which exploits them to improve instruction-level parallelism, reduce stack access latencies, reduce demand on the first-level cache, and reduce data bus traffic. Our results show that the SVF can improve execution performance by 29 to 65% while reducing overhead traffic for the stack region by many orders of magnitude over cache structures of the same size.

1. INTRODUCTION

In order to achieve ever higher performance in microprocessors, we continue to see an increase in complexity of the microarchitectural designs. To help manage this complexity and achieve designs that function in more restrictive time constraints, processor architects have relied more heavily on a technique of subdividing general structures into multiple, more specialized structures, which can be implemented more effectively. Examples include predicting indirect branches using a dedicated history buffer [9] and speculatively processing loads with good value locality [16]. Memory accesses can be further partitioned into address regions; examples of such regions include the instruction code, literal pool, static data, stack and heap regions. This partitioning can then be exploited to organize the cache structure to improve performance. Separate instruction and data caches are found in almost all processors, and recently we have seen architectures proposed that include stack and non-stack caches [10][11], as well as temporal and non-temporal caches [17]. Each of these designs uses a conventional cache organization and achieves improved performance by enabling parallel accesses to two cache structures and/or reducing contention in cache line allocation.

In this paper we focus on optimizing the performance of the stack memory references. We propose a structure called a stack value file (SVF), which is used to exploit the unique characteristics of stack references. The SVF is a non-architected register file containing the data near the top of stack, which would otherwise be held in memory or the data cache(s). All references to the stack are diverted to the SVF instead of the L1 data cache.

The contributions of this research are threefold. First, we perform a detailed evaluation of stack reference characteristics, including how stack access patterns interact with conventional cache structures in a sub-optimal manner. We then propose a new microarchitectural extension (the SVF) tailored to the reference patterns found in stack accesses. Finally, we evaluate the performance of our scheme in comparison to conventional cache designs and partitioned cache structures.

In Section 2, we examine the reference behavior of stack accesses and motivate the SVF design. Implementation issues and details of the SVF design are then provided in Section 3. Section 4 describes the experimental approach and benchmarks used. Experiments are then presented in Section 5 to show the performance potential of SVF. This work is contrasted with related approaches in Section 6. Section 7 presents conclusions and future work.

2. STACK REFERENCES

Memory accesses fall into several different categories, according to the region of memory they access and the access method used. The Compaq Alpha processor allocates memory space for the stack, growing from a system-defined virtual address down down towards virtual address 0. The location of the top of stack (TOS) is stored in the stack pointer (SP) register and identifies the lowest virtual address that may contain valid stack data. The middle address range above the stack is allocated during compilation and includes the read-only data region (.rdata), the code region (.text), and the global data region (.data). Dynamically allocated memory is placed in the heap, which grows upward from just after the global data region.

With the Compaq Alpha Processor linking and OS register conventions, the stack may be accessed by several means: via the stack pointer, *$sp*, the frame pointer, *$fp*, or through a general purpose register, denoted as *$gpr*.

Figure 1 shows the breakdown of memory references by region and access method for the SPEC CPU2000 integer

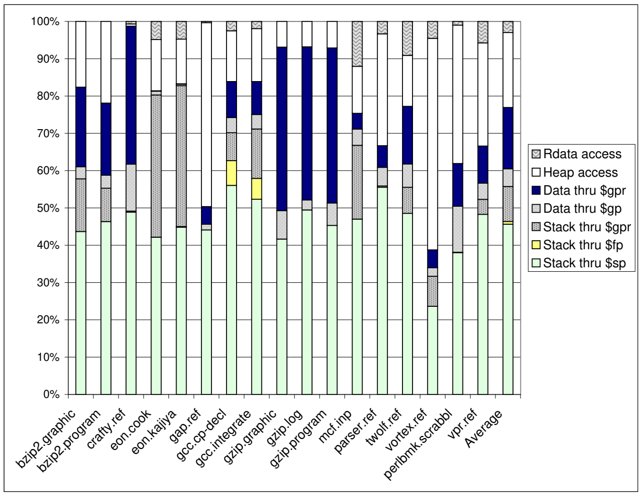

Figure 1: Run-time Memory Access Distribution for SPECint2000

benchmarks on an Alpha processor. This data is normalized to the total number of memory instructions for each application. For these applications, an average of 42% of the instructions executed access memory. Stack references account for an average of 56% of all memory accesses, while global data references account for only about 21% and a majority of the remaining accesses are to the heap. $sp-relative addressing is the dominant access method to the stack, accounting for 82% of all accesses to the stack, or 46% of total memory accesses. 252.eon is the single exception: over 45% of its stack accesses are performed using a $gpr. Since stack references are so common and $sp-relative addressing is the dominant access method to the stack, a closer examination of the characteristics of $sp-relative accesses is in order.

$sp-relative accesses are easily identified during instruction decode. Therefore they can be diverted, very early in the pipeline, to specialized processing units. Removing stack references from the general stream of references to the L1 cache reduces the demand for L1 cache bandwidth and may enable a reduction in the required portedness, size and associativity of the L1 cache. Processing stack references in parallel with conventional L1 cache references also increases the effective memory bandwidth, enabling more instruction-level parallelism. This has been demonstrated for stack machine architectures in the early CRISP processor [6] and more recently for conventional architectures in the stack cache [11].

Since $sp-relative addressing is simple and fast, the additional pipeline stage often used for complex address calculations can be avoided, enabling a shorter access latency. This early address calculation is easily performed for those references using $*sp*-relative addressing by using the techniques described in [3], and [5]. While accesses to locations in a special stack structure using methods other than $sp-relative addressing (e.g. with pointer accesses) must be handled, their low relative frequency enables higher latency access methods to be used without causing a significant performance penalty to the application.

Stack adjustments carry with them semantic assumptions regarding liveness that can be exploited to significantly reduce total memory bandwidth. A large fraction of the transactions between a stack structure or first-level cache and the second-level cache or main memory can be eliminated by leveraging this semantic information. Loads caused by writes to newly-allocated stack space can be avoided since any data fetched is by definition uninitialized and will be first accessed with a store. Writebacks of dirty lines that are in the region of memory that has been deallocated from the stack can also be eliminated since these memory locations will not be accessed until the stack space is reallocated with new data.

The SVF is able to rename stack addresses in the same manner as register renaming logic renames register space, eliminating anti-dependencies. Stack references are easier to rename for two reasons. First, the association of memory references with locations is simple and fast: the least significant bits of the address generated by adding the stack pointer and offset are used to directly index into the SVF. No associative lookups are required. Second, the choice of which references to rename is simple, namely the top N locations on the stack. No prediction of locality is required. In addition, the implemented register renaming logic for an out-of-order microarchitecture can be reused for stack reference renaming; thus minimizing the extra hardware required.

The first notable feature of stack references is that the working set for almost all applications consists of a single, contiguous address region. This enables the stack data to be conveniently stored in a simple, fast structure, without lots of tags and complex lookup procedures while retaining high access (hit) rate. Changes in working set addresses are also easily tracked by tracking explicit changes in the stack pointer register. Decreasing $sp adds new addresses to the expected working set, which increasing $sp not only eliminates those locations from the workspace of the application (temporarily), but more importantly *kills* the deallocated values, i.e., it guarantees that the next access is a store.

The next notable feature of stack use is the relative lack of variation in the stack address region. The top of stack (TOS) is adjusted at least twice for each procedure invocation (function call and return stack adjustments) and perhaps more often. However, these changes exactly cancel each other, leaving the TOS unchanged after the return. This leaves the working address range of the stack determined by the call depth of the application. For non-recursive applications this is quite limited.

Data was collected for SPECint2000 benchmarks using Alpha binaries to show how the stack depth varies over the lifetime of each program. The TOS address, relative to the stack base address, was logged each time the stack pointer is updated. Figure 2 shows graphs that map stack depth variation over time. The x-axis is execution cycles, starting at the beginning of the program. The y-axis plots the stack depth, starting from zero (stack base). The basic data size in the SVF is 64 bits of data[1], and this is the unit of the y-axis, so 1000 units corresponds to 8KB.

[1]Alpha is a 64-bit architecture.

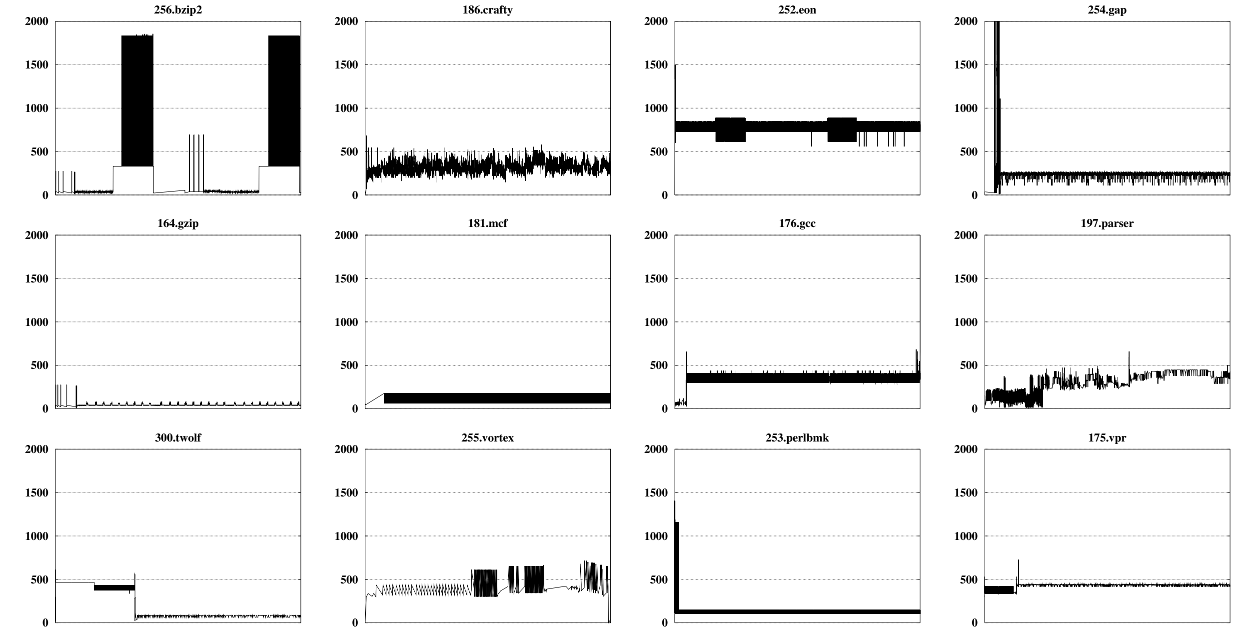

Figure 2: Examples of Stack Depth Variations (1 billion instructions)

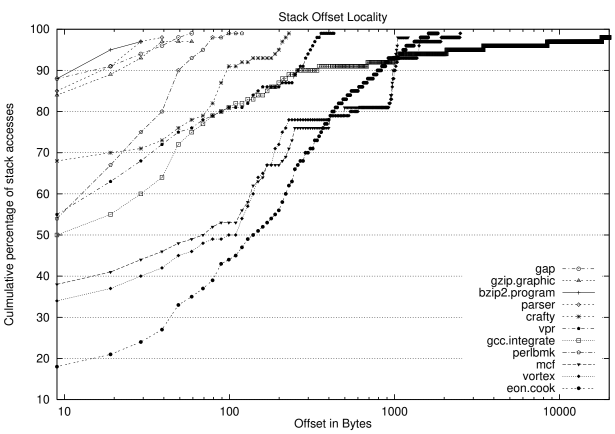

Figure 3: Offset Locality within a Function

There are two observations to be made from this data. First, a cache of 1000 units is larger than the maximum stack size for most of the applications. Even though it seems that 256.bzip2 has some variations larger than 1000 units, we show later that these variations incur very little overhead. Second, the stack depth is quite stable after the initialization phase, so the stack addresses mapped to the cache rarely change over the life of the applications. For example, in 186.crafty, the most representative active stack region is [200, 600], or about 400 units, requiring only a simple (even contiguous!) cache structure to capture almost all accesses.

Stack references also tend to be reasonably close to the top of the stack. Figure 3 shows the cumulative distribution within a function of offsets into the stack region (with the x-axis plotted on a log_{10} scale). Across all of the SPECint2000 benchmarks, the average distance from TOS for a stack reference ranges from 2.5 bytes (256.bzip2) to 380 bytes (176.gcc), and over 99% of all references (except for 176.gcc) are within 8KB of the TOS. Thus spatial locality with respect to the TOS is excellent. No references are beyond the top of the stack for these benchmarks. The graphs illustrate that most stack references are in one contiguous space (between 0 and 300 bytes offset from TOS), indicating that there is no need for a mechanism flexible enough to maintain a non-contiguous working set.

The conclusion to be drawn from this data is that an SVF that is 8KB or less will still capture almost all the stack references. While this may fail if the the variation in stack depth is large, the data presented in this subsection and later, in Section 5, suggests that potential losses due to managing only a single contiguous region at the top of the stack, contrasted with a more conventional cache allocation, are minimal if the SVF is adequately sized.

3. STACK VALUE FILE DESIGN

The stack value file is specifically tailored to optimize the storage of stack references. The SVF is a register file large enough to hold those stack locations near the TOS. Arrays with thousands of registers have become reasonable to build today. The SVF can be more area efficient than a standard cache design since it needs almost no tag space and can be direct-mapped instead of associative. It may also avoid the need for dual-porting the first-level cache.

The SVF is architecturally invisible, leaving the designer with the freedom to choose an appropriate level of support for stack references without the constraints of a large architected register file. References to cacheable locations allocated in the address range covered by the SVF are diverted from the first-level cache as described in Section 3.1, thus reducing its bandwidth, capacity and associativity demands. *$sp*-relative references are recognized early enough to avoid the added latency that general address calculations require. SVF references are renamed like general-purpose registers through the register alias table, further reducing delays and effectively implementing data forwarding. Figure 4 shows our re-architected pipeline. Other

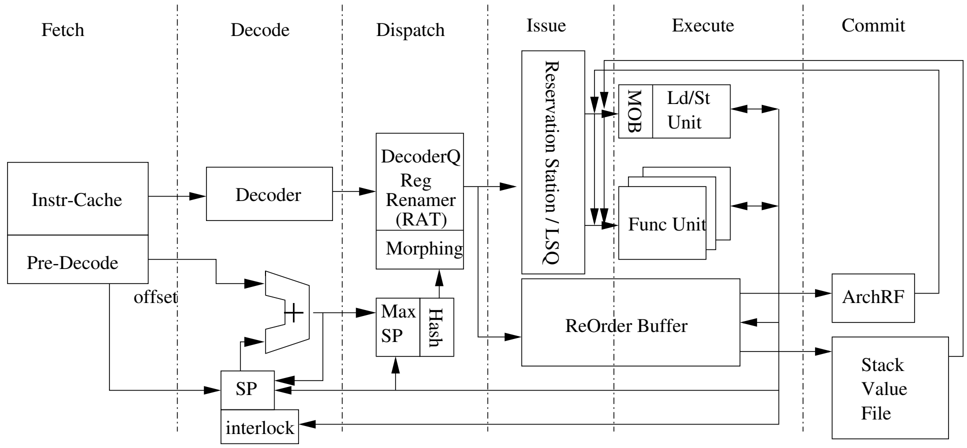

Figure 4: Microarchitecture Extension with a Stack Value File Implementation

references to locations in the SVF are detected using a bounds check. Such references are diverted from the 1st-level cache at a modest performance penalty, as described in Section 3.2.

The SVF is a circular buffer, with memory locations mapped to SVF registers according to the lowest-order address bits. Changes to the stack pointer are detected, and lead to data movement to and from the first-level cache as necessary. Status bits are associated with entries in the SVF to identify dirty data that needs to be pushed onto the stack. Valid and dirty bits minimize or delay traffic when a TOS adjustment or a SVF access miss makes it necessary to update the range of addresses mapped to the SVF. Only one tag is needed for each of the pages that the SVF spans. For example, an 8KB SVF needs only three tags for 4KB pages.

The following subsections describe these new microarchitectural features in greater depth.

3.1 Morphing Stack-Pointer Based References

An extended pre-decode circuit in the fetch stage is used to identify stack pointer-based memory references and to determine their immediate offset values. A special adder in the decode stage enables fast address calculation using this predecoded information. Prior studies [3][4] have already demonstrated that similar techniques are viable.

In our design, a pipeline interlock incorporated in the decode stage stalls instruction decoding if the stack pointer is updated in an unexpected manner (i.e. other than increments or decrements by an immediate constant value); since almost all $\$sp$ updates are simply adjustments with an immediate constant, we can perform these computations in the decode stage early by keeping a speculative $\$sp$ register copy in the decode stage. All subsequent $\$sp$-based references fetch $\$sp$ content from this speculative copy to index their SVF register ID. If the branch is correctly speculated, execution continues. However, if the branch is mispredicted, then the speculative $\$sp$ copy will be recovered with the value from the architectural $\$sp$ before the pipeline restarts at the correct branch path. All other $\$sp$ updates require a reference to other general purpose registers (except for zero register $r31 in Alpha). In those cases, the interlock stalls decoding to prevent following instructions from reading a stale TOS address.

Once a memory address with $\pm$IMM($\$sp$) addressing mode is computed in the decode stage, the address is checked against the range of stack memory currently held in the SVF. Clever implementations can even do the range check in parallel with the add. If a hit is detected, the instruction is morphed into a register-move operation and dispatched to the reservation station. The low-order bits of the address are used as the register ID to index into the SVF. These architecturally-transparent register IDs can be considered as an extension of general-purpose register IDs. The hardware register renamer can then rename each active SVF register into a corresponding entry in the physical registers of the reorder buffer. After the dispatch stage, all the morphed $\$sp$ load/store instructions will have been mapped into register space. Thus dependencies on these SVF registers are treated just like any regular register dependency.

3.2 Stack Memory Reference Disambiguation

Since only memory references indexed by the stack pointer are steered to the SVF, stack data references through other means, such as the frame pointer or general purpose registers, must be disambiguated and redirected into the SVF for data consistency.

For each stack-pointer based reference morphed into the register move form, two micro operations (uops) are generated after instruction decoding. One uop is the constructed register move (this may be able to be eliminated with renaming for a store [14]), while the other uop, computing the early resolved stack address, is enqueued into the Load/Store Queue (LSQ). The uop in the LSQ is used for disambiguation before the morphed references are committed to the SVF. If any later load instruction collides with these uops in the LSQ, as detected by the Memory Order Buffer (MOB), regular store forwarding will be performed.

All memory instructions that reference the stack memory region through registers other than the stack pointer have their addresses checked against the current stack range in the SVF. The load/store operation is then re-routed to

SVF if a match is detected.

There is one particular circumstance in which a simple re-routing operation cannot correctly maintain data dependency. This happens because of the relative timing of when references are determined to access the SVF. When a store through a general-purpose register is followed by a colliding load through stack pointer, the load can retrieve a stale value from the SVF (since the SVF access for the load occurs earlier in the pipeline). This condition is detected in the LSQ when the store executes. A pipeline squash, similar to the recovery from a memory ordering violation, is invoked to avoid a chain of incorrect data dependent instructions. During re-execution, this problem can be avoided by introducing a redundant dependence that forces the load to be executed late. The $*sp*-relative load is broken into two instructions. The first, designed to go through the execute stage, performs the address calculation ($*sp* plus offset). The second, dependent on the first, performs the load itself, at a time which is guaranteed to be after the store.

3.3 SVF Status Bits

Each SVF register contains two status bits. The dirty bit identifies the subset of all locations between the new and old TOS that need to be written out upon a TOS adjustment to maintain data coherence. The dirty bit is set when its corresponding SVF register is written, and cleared when the data is written back. The valid bit indicates whether locations exposed by a TOS adjustment need to be read. The valid bit is set when data is written in the SVF, either upon a write from the processor core or a fill from memory. It is cleared for locations between the new and old TOS when the TOS is adjusted (both shrinking and growing). If an SVF load accesses an invalid register, a load of that element is performed.

These status bits improve performance for the SVF design in several ways. First, the dirty bits avoid writing back clean data. Second, the valid bits avoid a burst of unnecessary reads when the stack shrinks. Locations are read only when needed, like a cache.

The granularity of these status bits is most naturally the smallest data type that is frequently used. For the Alpha architecture, this is 64 bits. If the granularity is larger than this, there will be more memory traffic.

At first glance a caching scheme may seem more advantageous than a contiguous scheme out of concern for penalties on context switches, but this need not be an issue. The use of a stack does not necessarily eliminate writebacks of dirty data on a context switch. Any new process is likely to displace much of the data in their stack cache. In fact, per-word valid bits used in the SVF reduce the traffic over a conventional cache during a context switch. If shown to be necessary because of localized poor SVF performance, the SVF can be dynamically disabled for a period of time.

4. EXPERIMENTAL APPROACH

The SPECint2000 benchmark programs are used in this study. The binaries were compiled using the Compaq Alpha compiler with appropriate optimizations enabled. The input files are either from reference input set or training input set, as shown in Table 1.

Benchmark	Input
256.bzip2	ref: graphics & program
186.crafty	ref: crafty.in
252.eon	cook & kajiya algorithms
254.gap	ref.in
176.gcc	train: cp-decl.i & ref: integrate.in
164.gzip	ref: graphic & program & log
181.mcf	ref: inp.in
197.parser	ref.in
300.twolf	ref
255.vortex	ref
253.perlbmk	train: scrabbl.in
175.vpr	ref

Table 1: SPEC CPU2000 integer benchmark

The simulators used in this research were derived from the *SimpleScalar* [8] tool suite, an execution-driven, cycle-accurate, out-of-order superscalar processor simulator. The processor simulated adopts a Register Update Unit (RUU) [19] approach that combines the functionality of a Reservation Station (RS) and a ReOrder Buffer (ROB). We modified the pipeline to incorporate our stack value file design.

Components	4-wide	8-wide	16-wide
Decode width	4	8	16
Issue width	4	8	16
Commit width	4	8	16
IFQ size	16	32	64
RUU size	64	128	256
LSQ size	32	64	128
IL1 cache	8-way 256KB	8-way 256KB	8-way 256KB
DL1 cache	4-way 64KB	4-way 64KB	4-way 64KB
IL1 hit	1 cpu clk	1 cpu clk	1 cpu clk
DL1 hit	3 cpu clks	3 cpu clks	3 cpu clks
Unified L2	4-way 512KB	4-way 512KB	4-way 512KB
L2 hit	16 cpu clks	16 cpu clks	16 cpu clks
Mem latency	60 cpu clks	60 cpu clks	60 cpu clks
CPU-Mem clk ratio	6:1	6:1	6:1
Store forwarding	3 clks	3 clks	3 clks
Int/FP ALU	16	16	16
Int/FP Mult	4	4	4

Table 2: Processor Models.

The machine models used in our experiments are summarized in Table 2. It is worth noting that the store forwarding latency used in all of our experiments is 3 cycles. As a result, the L1 cache hit latency is also 3 cycles. These match our measurements of the actual latency on the Intel Pentium III processor, including cache latency and store forwarding delay in the pipeline. L1 cache accesses are fully pipelined.

In order to demonstrate the performance potential of our scheme and to reduce the performance interference from the front-end, we use a perfect branch predictor as well as a fairly large and fast first-level instruction cache.

5. PERFORMANCE EVALUATION

In this section, we provide data to show how the characteristics of stack references can be effectively exploited with a stack value file to alleviate L1 cache bandwidth congestion, reduce latency for stack memory references, reduce the memory traffic for the memory subsystem, and ultimately improve execution performance.

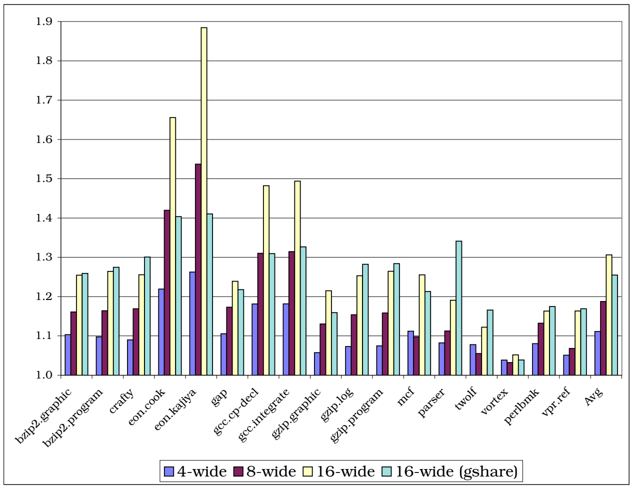

Figure 5: Speedup Potentials of Morphing All Stack Accesses to Register Moves

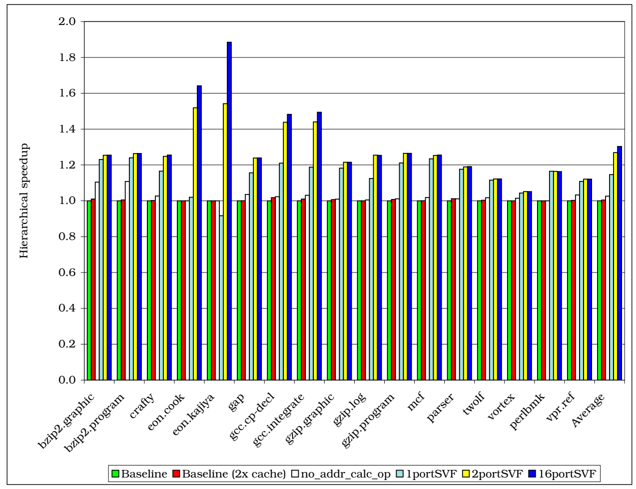

Figure 6: Progressive Performance Analysis

5.1 Improving Cache Bandwidth, Latency and ILP

The primary benefits of treating stack references separately from all other memory references are the opportunities the SVF provides for:

- exploiting more instruction-level parallelism with the existing physical registers in the RUU and additional SVF ports for stack references
- disambiguating stack references through existing register alias table
- eliminating the stack references on the data cache ports

The latency of the stack references can be reduced if the SVF entries can be accessed like registers. The gains from these improvements are quantified in Figure 5. This figure demonstrates the potential performance gains from implementing an SVF with infinite size and SVF ports for various generations of processors, assuming all the stack references can be morphed into register-to-SVF moves. The first three bars show average speedups of 11%, 19%, and 31% for a 4-wide, 8-wide and 16-wide machines respectively, with a dual-ported first level data cache and a perfect branch predictor.

The 4-, 8- and 16-wide speedups are all relative to a baseline with a perfect branch predictor. The last column shows 16-wide speedups with gshare, relative to a baseline with gshare. The average speedup is 25%. Some cases show a greater speedup with gshare. In these cases, SVF's latency-shortening benefits allow branches to be resolved early, reducing the branch misprediction penalty. However, the more realistic branch prediction of gshare reduces the effective basic block size, reducing the potential stack parallelism and leading to a smaller average gain than for the perfect prediction case.

5.2 Progressive Performance Analysis

To understand the performance gain in a quantitative manner, Figure 6 shows progressive performance improvements under different constraints for a 16-wide machine. Starting from the baseline machine model described in Table 2, we relax the machine constraints for each bar in the figure.

First, the L1 data cache size is doubled (from 64KB to 128KB[2]) without increasing the access latency. As shown in Figure 6, the speedups from enlarging L1 size for all the SPECint2000 benchmarks are negligible. In the next configuration, we remove address computation instructions for all stack references (denoted as no_addr_cal_op in the graph), thereby eliminating their dependencies. This dependency reduction benefits some benchmarks such as 256.bzip2, which improves by 11%. Since our processor model supports out-of-order execution with a 256-entry RUU, the address calculation can be easily hidden by other independent instructions, and the overall speedup is only 3%. This observation concurs with the results reported in [4] where their zero-cycle load technique posted significant gains only for in-order machines.

Most of the performance boost comes from the implementation of the stack value file, posting an additional improvement of 28% for a 16-port SVF. We show the speedups of a SVF with 1, 2 and 16 ports. A dual-ported SVF on a 16-wide machine (which gives an incremental gain of 27%) performs almost on par with a 16-ported SVF for most of the SPECint2000 benchmarks. This suggests that a limited number of ports covers most of the potential gain, except for 252.eon and 176.gcc. These two benchmarks seem to have many more clustered stack references that lie on the critical path of the performance. A larger number of SVF ports accommodates this bursty stack reference parallelism.

5.3 SVF vs. Stack Cache

5.3.1 *Performance*

A related approach, the decoupled stack cache [11], is compared against our SVF scheme along with the baseline

[2]For the rest of the configurations in Figure 6, the L1 data cache size remains at 64KB.

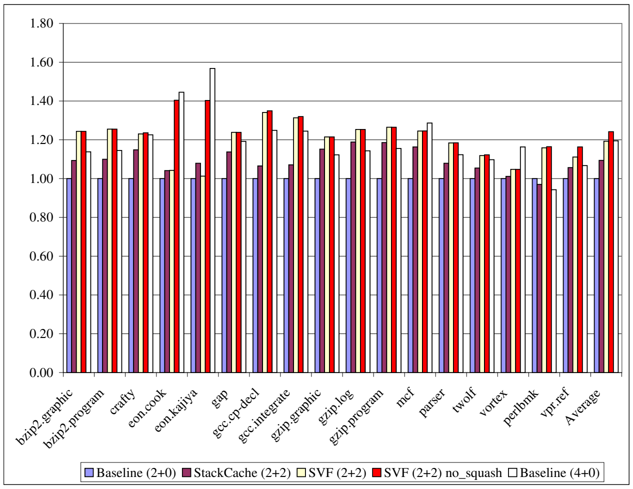

Figure 7: Comparison of different cache implementations

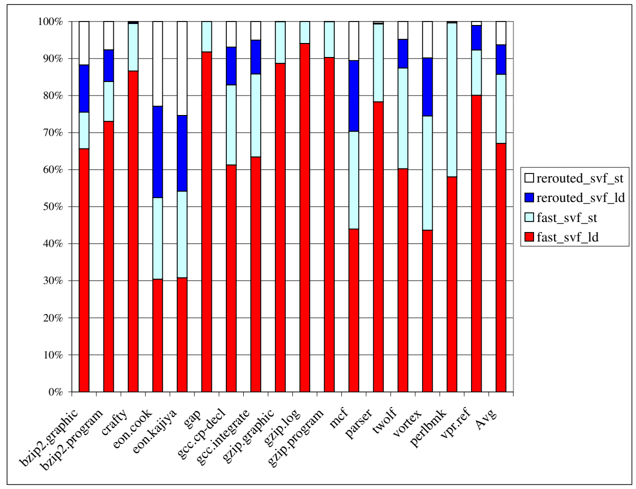

Figure 8: Breakdown of SVF Reference Types

microarchitecture. The stack cache is implemented as a direct-mapped cache and has the same capacity (8KB) as our stack value file (1024 entries × 8 bytes). Figure 7 shows the comparison of the SVF, stack cache and baseline approach with different port combinations. The *(R+S)* symbol represents the configuration with *"R"* regular L1 cache ports and *"S"* SVF or stack cache ports. The (4+0) configuration uses a longer data cache hit latency (4 cycles instead of 3 cycles) than (2+S)'s because of the larger number of ports. The performance numbers of the SVF scheme were generated by an actual implementation described in Section 3.

The baseline (4+0) might be expected to outperform the (2+2) because the four universal L1 data cache ports in (4+0) can service four concurrent memory references, no matter which memory regions these references are going to, whereas two memory references out of the (2+2) must be from stack, otherwise the ports are left unused. However, in several cases, the SVF scheme outperforms the more flexible configuration, yielding a 4% improvement overall. One reason is the longer latency in (4+0). The other is that input data of instructions on the critical path can be directly read from the physical registers in the RUU, indexed through the register alias table. Even though the store-forwarding mechanism exists in a conventional microarchitecture, yet it will take some cycles (3 in our case) to poll for a hit in the LSQ.

There is one anomaly, 253.perlbmk, where the stack cache (2+2) runs a little bit slower than the baseline (2+0). We found that the stack cache misses dominate the critical path and the working set of stack data does not fit into the stack cache although it fits into L1 cache for the baseline architecture.

Figure 7 also shows that the (2+2) SVF implementation outperforms the (2+2) stack cache scheme with one exception, 252.eon. In eon, we found that a large number of load squashes occur due to stores through $gpr followed by loads through $sp where these references map to the same stack addresses. As discussed in Section 3.2, these squashing activities can be eliminated using a different code generator tailored for the SVF implementation. By applying this optimization, represented by the *no_squash* bars, we can greatly improve the performance of the 252.eon, making it outperform the stack cache scheme by over 30%. Without the no_squash code optimization, the SVF outperforms the stack cache by roughly 9%, and with the no_squash feature the average is 14%.

Figure 8 shows the breakdown of the SVF references. The fast SVF loads and stores are the references that were directly morphed in the front-end pipeline. References that are not $sp-relative need to be rerouted into the SVF after their addresses are calculated. On average, around 86% of stack references can be directly morphed into register moves in the front-end, while 14% of them are re-routed into the SVF.

5.3.2 Memory Traffic

The main performance difference between the SVF and a stack cache arises from the exploitation of the semantic information inherent in adjustments to the stack pointer. Because the region of memory contained in the SVF is guaranteed to be contiguous, some assumptions can be made that cannot be made for a stack cache:

1. Allocations: A new allocation made as the stack grows down for a SVF implies that the data must be invalid. No such assumption can be made for a stack cache, since the data may have already been written and replaced. Thus a stack cache must read the rest of the line before data can be written.
2. Dirty Replacements: When locations are replaced as the stack shrinks for the SVF, they are semantically guaranteed to be dead, and need not be written back. No such assumption can be made for a stack cache, and the line must be written back.

Table 3 summarizes the in (read) and out (write) memory traffic incurred for our SVF design and a stack cache design, for different SVF and cache sizes. The stack cache's

size	*2KB*				*4KB*				*8KB*			
	Quad-Words In		Quad-Words Out		Quad-Words In		Quad-Words Out		Quad-Words In		Quad-Words Out	
Benchmark	Stack $	SVF	Stack $	SVF	Stack $	SVF	Stack $	SVF	Stack $	SVF	Stack $	SVF
bzip2.graphic	1350700	41	493604	14355	452848	3	101124	14333	744	3	0	0
bzip2.program	770608	44	289096	8558	434992	3	109684	8536	756	3	0	0
crafty.ref	45811024	5	6634372	31	572	5	72	0	556	0	0	0
eon.cook	85730448	0	37119416	101	677452	0	16180	32	1568	0	532	0
eon.kajiya	33019440	0	23431164	101	12512884	0	11544940	32	905544	0	446440	0
gap.ref	193148	66	175880	4905	130148	115	125564	5832	118232	48	116344	0
gcc.cp-decl	17523908	520	11472364	85939	13323984	520	8756656	30949	6783056	520	5274116	1177
gcc.integrate	30826224	91	20421952	56969	27329584	91	18086172	22152	22657988	91	15272932	2382
gzip.graphic	296	38	176	0	144	0	0	0	144	0	0	0
gzip.log	296	38	176	0	144	0	0	0	144	0	0	0
gzip.program	324	41	200	0	148	0	0	0	148	0	0	0
mcf.inp	220	19	40	0	204	0	0	0	204	0	0	0
parser.ref	76980	15875	76108	13	592	0	80	0	592	0	0	0
twolf.ref	2989324	233	2762292	0	652	25	188	0	564	0	0	0
vortex.ref	988	53	712	0	488	53	44	0	480	0	0	0
perlbmk.scrabbl	99116	55	89508	49377	89948	36	81112	49429	79312	36	78812	0
vpr.ref	1432	0	1104	0	652	0	152	0	644	0	0	0

Table 3: Memory Traffic for Stack Cache and SVF schemes

memory traffic is caused by compulsory, capacity, and conflict misses, along with dirty writebacks, which generate traffic between the stack cache and the L2. The SVF's traffic to the L1 only occurs on demand, for dirty and live data. For instance in 256.bzip2 with a 2KB stack cache, about 1.35 million quad-words were allocated into the stack cache and 0.49 million quad-words were evicted due to dirty replacements. In contrast, the SVF read in only 41 quad-words and wrote out 14,355 quad-words. In most of the scenarios, the SVF dramatically reduces traffic by many orders of magnitude. As aforementioned, this is because the SVF transfers dirty data in a finer granularity and requires no load on write misses. In addition, the SVF does not write the deallocated stack frame out to memory as data on deallocated stack frame are semantically dead.

5.3.3 *Context Switches*

Upon a context switch, there is likely to be an increase in memory traffic for either a stack cache or the SVF, because both have to write back dirty data as the new process displaces the current process's data. Since both the stack cache and the stack value file are small, a large percentage of their locations are likely to be replaced soon after the context switch. One might then anticipate significant writeback traffic soon after the context switched occurred.

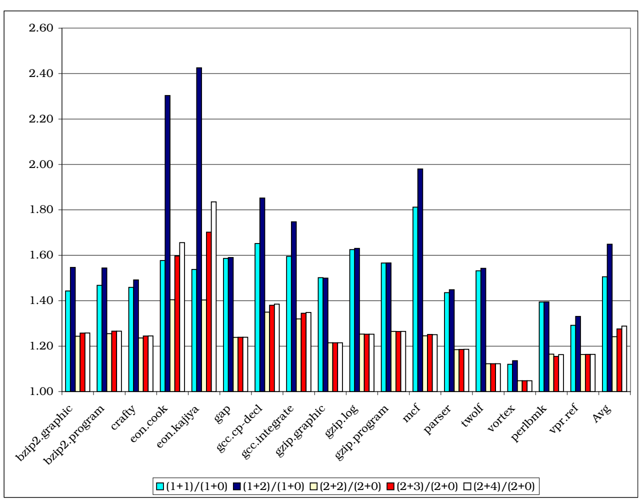

Figure 9: Performance Improvements over Baseline Microarchitecture

The SVF has less traffic to the L1 on a context switch than a stack cache has to the L2. This is because the SVF invalidates deallocated stack frames, leaving fewer valid and dirty bytes to write back. Furthermore, the SVF's dirty bits have a finer granularity than those of the stack cache. The SVF's contiguity eliminates the need for a tag on each line, and more dirty bits can be used instead. Thus while a stack cache needs to write back a whole line when one word is dirty, the SVF writes back only a 64-bit block. Table 4 quantifies the bytes of traffic for both stack cache and stack value file, averaged over total number of context switches with the context switch period of 400000 instructions. Writeback traffic for the stack cache in the case of eon is about 7000 bytes per context switch on average, which is about 10 times more then in the case of stack value file. The table illustrates that write back traffic for the stack value file is 3 to 20 times smaller than the writeback traffic for the stack cache.

Benchmark	Stack Cache	Stack Value File
256.bzip2	83	33
186.crafty	1040	201
252.eon	7053	740
254.gap	830	64
176.gcc	4235	188
164.gzip	3	1
181.mcf	477	153
197.parser	1253	66
300.twolf	727	248
255.vortex	7	2
253.perlbmk	571	134
175.vpr	800	87

Table 4: Memory Traffic on Context Switches

5.4 SVF Performance

Finally, the speedups measured for the actual SVF implementation versus a baseline machine are illustrated in Figure 9. The execution speedup measured for both single-ported and dual-ported data caches are shown. The average performance improvement for adding a single-ported SVF to a single-ported data cache is 50%. Most current processors incorporate dual-ported data cache designs. However, low-cost/low-power or embedded processor designs may leverage a large single-ported data cache with a small double-ported stack value file to reduce both cost and power dissipation while improving overall performance.

When the SVF is dual ported, improvement climbs to

65%. For most of the benchmarks, performance is saturated when two SVF ports are supported, except for 252.eon, which continues to improve its performance as the number of SVF ports are increased. Improvements are lower for cache designs supporting dual-ported first level data caches since port contention is reduced; however, adding an SVF still yields significant additional improvement. For a reasonable configuration with a dual-ported SVF added to a dual-ported data cache, performance improves by an average of 24% over a conventional microarchitecture with a maximum performance improvement of 84% for 252.eon.

6. RELATED WORK

Techniques for fast procedure calls [15] were broadly studied in the 1980's when CISC machines were still in the limelight. The overheads of saving and restoring the register file associated with each procedure call were rather significant [13]. Many prior commercial and research microprocessors had tried to address this issue at extra hardware cost.

The HP3000 Series II [7], a stack-oriented architecture designed by Hewlett-Packard in the late 70's, used a 4-entry top-of-stack (TOS) cache as an extension to stack memory. The CRISP [12] and Hobbit [2] processors, developed at Bell Labs adopted a complete memory-to-memory instruction set architecture with few addressing modes to avoid the overheads of procedure calls. The design offloads the burden of register allocation on the top of the stack from the compiler to the hardware by incorporating a small stack cache (32 entries). The processors index the stack cache on-the-fly using the low-order bits of the referenced address. The stack cache was the processors' only data cache.

Register windows [21] or the register stack engine (RSE) [1] are used in some of today's high-performance microprocessors to eliminate the overhead of procedure calls and returns. Extra instructions may be needed, e.g. *save* and *restore* in SPARC-V9 or *alloc* in IA-64. This general approach is part of the architecture, not just the implementation.

There have been several proposals for early address resolution to improve memory instruction latencies. These techniques enable our SVF design by providing a mechanism for early address resolution. Austin, Pnevmatikatos and Sohi [3] introduced a fast address resolution scheme by predicting effective addresses early in the pipeline. They found that with simple compiler and linker support, the prediction accuracy ranges from 62 to 99%. In [4], Austin and Sohi proposed and evaluated pipeline designs to support zero-cycle loads. Although the speedups are encouraging for in-order processors, the speedups for latency-tolerant out-of-order processors are generally less than 10%. In [5], Intel researchers proposed a technique dubbed *register tracking* for early memory address resolution for operations of the form reg±imm in the front-end pipeline. They demonstrated that this technique reduces load-to-use latencies to the data cache by experimenting a deep pipeline which contains 8 stages between decode and execution.

The number of cache ports becomes more crucial as processors' issue widths get wider and they have more aggressive and accurate multiple branch prediction mechanisms. In more recent work [11], Cho, Yew and Lee proposed a data-decoupled architecture that partitions memory references into different streams and feeds them through decoupled memory pipelines for execution. They studied the performance impact of decoupling local variables allocated on the run-time stack [11]. They concluded that a small 2KB local variable cache (LVC) achieves a 99% hit rate for most of the SPEC95 benchmark programs and as a result, leave more headroom for data cache bandwidth. In their follow-up work [10], they introduced the notion of access region locality and proposed an access region prediction table (ARPT) in the fetch stage to predict which region an instruction is referencing. In the decode stage, the memory operation is directed into the predicted region pipeline for future processing.

Tyson and Austin [20] devised a mechanism that performs memory renaming dynamically to reduce memory traffic. A memory dependency predictor is used to predict the relationship of a producer (stores) and a consumer (loads). The predictor uses this information to index a non-architected value file for loads. They employ a confidence mechanism to control the prediction. Their simulation shows 16% performance improvement in average.

Rivers et al. [18] identified limitations of existing multi-ported cache designs and proposed a Locality-Based Interleaved Cache (LBIC) that multi-ports a line buffer instead of the entire cache bank to exploit the spatial locality of a cache line while reducing the cost of building a true multi-ported cache.

7. CONCLUSIONS

In this paper, we perform a detailed analysis of stack reference behavior identifying several unique characteristics in how the stack is accessed. These characteristics led us to propose a new microarchitectural enhancement, the stack value file, designed to optimize the stack references induced by high-level language conventions.

The contributions of this research are threefold:

1. We identify several characteristics of stack references that differ from general data references. These include: a single contiguous access region (eliminating the need for tags), a much higher percentage of first reference store operations (making per word valid bits attractive), frame deallocation invalidates dirty data above the new TOS (making writebacks unnecessary), and most references use a single $sp-relative address mode (making fast address calculation feasible).
2. We propose a new microarchitectural structure, the SVF, to exploit those characteristics and show how it can be integrated into existing processor pipelines to improve cache access latency and reduce memory traffic requirements.
3. We evaluate our scheme, comparing it to the best-performing previous cache-oriented approaches to partitioning stack references. These results show that a

SVF can obtain a 24% average improvement for conventional microarchitectures, while significantly reducing memory overhead traffic over split stack/non-stack caches.

Furthermore, our microarchitecture design transforms stack-pointer based memory accesses into moves between registers. This increases exploitable instruction-level parallelism by adding ports, off-loading bandwidth from the first-level data cache, and reducing the latency of the access. For a 16-wide machine, this increases performance by an average of 31% for an SVF of infinite size and ports for the SPECint2000 benchmarks.

Overall, these performance results make the stack value file an attractive design option, boosting performance without significant increases in area or complexity. The die area allocated to the SVF can be reallocated from space that otherwise would've gone to a larger first-level cache. The SVF is direct-mapped, can be single-ported, and can easily be banked. It uses almost no tag area, unlike its cache counterpart.

The additional complexity for the SVF is quite limited. \$*sp*-relative stack references are identified easily, and require little special handling. References to the stack without an \$*sp*-relative addressing mode are infrequent, so the added recovery cost is reasonably amortized. Cache tag space is eliminated in preference to a per word valid bit, resulting in little or no additional data storage overhead relative to a stack cache implementation.

For a deeper pipelined processors, our technique should deliver increasing performance gain as the value of early address computation is increased. Our next research project will be to extend this analysis to the x86 architecture with its increased reliance on the stack region and its use of partial word references.

8. ACKNOWLEDGEMENTS

This work was supported by NSF Career Grant MP-9734023, an Intel Foundation Fellowship and grants from IBM.

9. REFERENCES

[1] IA-64 Application Developer's Architecture Guide. Intel Literature Centers, 1999.

[2] P.V. Argade, S. Aymeloglu, A.D. Berenbaum, M.V. DePaolis, R.T. Franzo, R.D. Freeman, D.A. Inglis, G. Komoriya, H. Lee, T.R. Little, G.A. MacDonald, H.R. McLellan, E.C. Morgan, H.Q. Pham, and G.D. Ronkin. Hobbit: A High-Performance, Low-Power Microprocessor. In *Proceedings of COMPCON Spring*, 1993.

[3] Todd M. Austin, Dionisios M. Pnevmatikatos, and Guri S. Sohi. Streamlining Data Cache Access with Fast Address Calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[4] Todd M. Austin and Guri S. Sohi. Zero-cycle Loads: Microarchitecture Support for Reducing Load Latency. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[5] Michael Bekerman, Adi Yoaz, Freddy Gabbay, Stephan Jourdan, Maxim Kalaev, and Ronny Ronen. Early Load Address Resolution Via Register Tracking. *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[6] A. D. Berenbaum, D. R. Ditzel, and H. R. McLellan. An Introduction to the CRISP Architecture. In *Proceedings of the Spring COMPCON*, 1987.

[7] Russell P. Blake. Exploring a Stack Architecture. *IEEE Computer Magazine*, May 1977.

[8] Doug C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin-Madison, 1997.

[9] Po-Yung Chang, Eric Hao, and Yale Patt. Target Prediction for Indirect Jumps. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 1997.

[10] S. Cho, P-C. Yew, and G. Lee. Access Region Locality for High-Bandwidth Processor Memory System Design. In *Proceedings of the 32nd International Symposium on Microarchitecture*, 1999.

[11] S. Cho, P-C. Yew, and G. Lee. Decoupling Local Variables Accesses in a Wide-Issue Superscalar Processors. In *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[12] David R. Ditzel and H. R. McLellan. Register Allocation for Free: The C Machine Stack Cache. In *Proceedings of 1st International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1982.

[13] Joel Emer and Doug Clark. A Characterization of Processor Performance in the VAX-11/780. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984.

[14] 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. In *Proceedings of 31st Annual International Symposium on Microarchitecture*, 1998.

[15] Butler W. Lampson. Fast Procedure Calls. In *Proceedings of 1st International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1982.

[16] Mikko H. Lipasti, Chris B. Wilkerson, and John Paul Shen. Value locality and local value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[17] Jude A. Rivers and Edward S. Davidson. Reducing Conflicts in Direct-mapped Caches with a Temporality-based Design. In *Proceedings of the 1996 International Conference on Parallel Processing*, 1996.

[18] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, and Todd M. Austin. On High-Bandwidth Data Cache Design for Multi-Issue Processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[19] Guri Sohi and Sriram Vajapeyam. Instruction Issue Logic for High-Performance Interruptable Pipelined Processors. *Proceedings of 14th Annual International Symposium on Computer Architecture*, 1987.

[20] Gary Tyson and Todd Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997.

[21] David L. Weaver and Tom Germond. *The SPARC Architecture Manual*. SPARC International, Inc., 1994.