Smart Register Files for High-Performance Microprocessors

Matthew A. Postiff and Trevor Mudge
Advanced Computer Architecture Laboratory, EECS Department
University of Michigan
Ann Arbor, MI 48109-2122
{postiffm, tnm}@eecs.umich.edu
June 28, 1999

Abstract

This report examines how the compiler can more efficiently use a large number of processor registers. The placement of data items into registers, called *register allocation*, is known to be one of the most important compiler optimizations for high-speed computers because registers are the fastest storage devices in the computer system. However, register allocation has been limited in scope because of aliasing in the memory system. To break this limitation and allow more data to be placed into registers, new compiler and microarchitecture support is needed.

We propose the modification of register access semantics to include an indirect access mode. We call this optimization the Smart Register File. The smart register file allows the relaxation of overly-conservative assumptions in the compiler by having the hardware provide support for aliased data items in processor registers. As a result, the compiler can allocate data from a larger pool of candidates than in a conventional system. An attendant advantage is that the smart register file reduces the number of load and store operations executed by the processor. The simple addition of an indirect register access mode not only simplifies alias handling, but also provides opportunities for other optimizations. This dissertation examines several such optimizations.

1 Introduction

The performance of the memory hierarchy has become the most critical element in the performance of desktop, workstation and server computer systems. This is primarily due to the growing gap between memory and processor speed. Memory latency has been decreasing by 7% per year while processor frequency has been increasing at a rate of 55% per year since 1986 [56]. This growing gap results in pipelines that are idle for as much as half of all cycles and CPIs no better than 1.5 on a two-issue superscalar processor [57]. Designers have found that using on-chip cache memory is one way to combat the memory problem. Putting caches on chip is possible because integration levels have been increasing according to Moore's law [58]. The result is processor chips with 128KB or more of on-chip L1 and L2 cache [66].

Another reason that memory performance is critical to computer speed is that about 35% of instructions in the typical instruction stream are memory operations [56]. This is true for both CISC and RISC architectures. Thus with only one port to memory, three instructions per cycle is about the most an architect could expect to execute.

This "memory problem" forms the motivation for this research. There has been much research on it already, some of which will be described in Section 2. This work focuses on the highest level in the memory hierarchy, the register file, and compiler algorithms to manage it effectively. The compiler has information about data access patterns and aliasing relationships that can direct optimizations to improve performance. In the case of the register file, it can do this in several ways:

1. Allow the use of more processor registers.
2. Reduce the number of load and store operations.
3. Specify prefetching information to hide memory latency.
4. Schedule loads above potential dependencies to further hide latency.

The compiler already allocates some data to registers but has little direct control over lower levels of the memory hierarchy. Consider the memory hierarchy for a typical high-performance desktop or server system today: 1) A small register file with many read and write ports (say 12R, 4W) which is only directly addressable by the compiler; 2) An L1 data cache, which has one or a very few ports but which is addressed by a runtime-computed address; 3) A similarly configured L2 cache with even fewer ports; 4) A single-ported main memory addressed by an arbitrary computed address.

In such a configuration, the gap between the access semantics of registers and those of L1 data cache is quite wide, both in number of ports and the flexibility of addressing. This report considers the "Smart Register File" (SRF) which is a register file which trades flexibility in access semantics for number of ports and access time. These registers are "smart" because they are co-managed by the hardware and the compiler.

More generally, a "Smart Short Term Memory" is any on-chip memory structure which is co-managed by the compiler and hardware. Typically, data resides for a short time in a short-term memory. As such, the short term memory is not the "home" location for the data [9]. An example of a short term memory is a compiler-managed zero-page which is referenced by base+index addressing mode. Such a memory structure is a compromise between the direct-only addressing of a register file and the computed address of L1 data cache. It is in the memory hierarchy so it can be co-managed by the hardware but

data can be assigned to the zero-page by the compiler. This particular example allows two of the benefits we enumerated above. First, the compiler can eliminate load and store instructions if there are specialized instructions in the ISA for accessing the zero-page. Second, the compiler can do some prefetching into the zero-page if it so desires. Another example is a hardware-managed L1 data cache with prefetch hints supplied by the compiler. It is "smart" because it attempts to combine the intelligence from the hardware (local dynamic or runtime knowledge) and from the compiler (global static knowledge) and apply it to the problem of memory latency.

For the remainder of this report, we discuss one possible design for a short term memory which we have not seen in the literature. We call this design the Smart Register File, or SRF for short.

1.1 The Smart Register File

The aim of this research is to develop ways to increase microprocessor performance by enabling more data to be stored in the registers. Specifically, we will demonstrate that a key problem is the aliasing problem, which we propose to solve with a combination of compiler and hardware innovations. The results presented in this research will show the potential performance increases.

The crux of our proposed solution is to employ some additional hardware associated with each register in the processor so that aliasing can be detected as the program is executing. By providing direct support in hardware for aliased data items in processor registers, overly-conservative compiler assumptions can be relaxed and a better allocation of data to registers can be obtained. This allows registers to be used more efficiently (when there are free registers). It may also increase register pressure to the point of requiring more registers in the architecture, but the SRF does not *require* a large number of registers.

The next subsection introduces the alias problem and how it restricts register allocation. The following subsections outline the benefits of the SRF; it will be described in detail in Section 3.

1.1.1 The Alias Problem

An alias is a condition where a datum is referenced through more than one name. The alias problem can be explained either from a hardware or compiler perspective. In the hardware, it is also called the memory disambiguation problem. The hardware would like to execute a load instruction as early as possible in order to allow dependent instructions to start. However, if there is a pending store already in the pipeline whose address is not known, the hardware must assume the worst–that the unknown address is the same as the address being requested by the load. Thus the load must wait for the store to finish execution. The memory disambiguation hardware enforces the program ordering on memory operations. The load and store are said to contain *ambiguous* memory references until the addresses are resolved.

Ambiguous memory references affect the compiler as well. Some rudimentary alias analysis is required of the compiler and most aggressive optimizations need advanced alias analysis to be correct and effective. Figure 1 shows some C code where the

4

```
int x;
int *p;
p = &x;
...
x = 1;
*p = *p + 2;
...
print x, *p
```

Figure 1. Code demonstrating the alias problem. The variable x cannot be allocated to a register because it's address is taken and it is potentially modified through the pointer p (The ellipses are unspecified code which could contain arbitrary control flow.) The load of *p for the addition cannot be executed before the initialization of x because it might get an out-of-date value.

alias problem becomes evident. Variable *p* points to *x*; thus the data in variable *x* has two names. Thus it cannot be stored in a processor register, which has only one name.

In first order alias analysis, the compiler can determine whether a variable can ever be aliased by whether its address is ever taken. If not, then the variable cannot be referred to through more than its own name and an alias cannot occur. However, the compiler cannot determine statically the values of *computed* addresses. Thus it often cannot be sure whether an alias could occur or not. It must assume the worst–that the computed addresses can refer to the same location. If it were to copy the value of a variable into a register and then the value of that variable was changed through a second name, the copy would be inconsistent with the actual value in memory.

More advanced alias analysis requires examination of the whole program, even across module boundaries, to determine if two addresses can refer to the same data.

The final result of all this is that the compiler must be conservative. It does this by leaving variables in memory and referring to them through load and store instructions. These extra load and store instructions are not always necessary, though, because the aliasing condition might not happen all the time. That is, the compiler assumed it would happen all the time but a certain percentage of the time (dynamically), there is no alias. It is these extra operations that we would like to remove.

The SRF allows the compiler to relax conservative assumptions and refer to aliased data through a register name. Thus more data can be stored in processor registers and existing register sets can be used more efficiently.

1.1.2 Using More Registers

Studies have shown that the number of high-speed registers that can be effectively used is limited to a few dozen [4, 62]. There are several reasons for these results. First, the average programmer does not keep track of more than a few variables in a function, and the number of temporaries used by the compiler is typically small.

In processors that support multiple instruction issue (both superscalar and VLIW/ EPIC styles) the availability of large numbers of registers is particularly important. With only a few registers, program performance can be limited by spill instructions inserted by the compiler to deal with the small number of registers. These additional instructions can

nullify the benefit of multiple issue as the extra issue slots are used for the data movement operations instead of operations directly related to the algorithm.

For certain classes of benchmarks, aggressive loop unrolling, software pipelining (and modulo variable expansion), unroll-and-jam, and inter-procedural optimizations such as inlining can significantly increase the number of registers required. Even the traditional optimizations such as copy propagation, common subexpression elimination, induction variable elimination, and code hoisting increase register pressure by adding temporary scalar values and extending scalar lifetimes [16, 36].

Furthermore, whole classes of variables are ignored in most register allocation studies (e.g. those referenced above). sGlobal variables are not usually allocated to registers, even though they may be able to reside in a register for their entire lifetime. There are significant numbers of global variables that fall into this category, as will be shown later in this report. Structure or array elements are also not usually placed into processor registers.

Finally, aliased variables are not considered as candidates for register allocation either because of the alias problem. As will be shown, these variables are responsible for a significant number of memory accesses.

Aggressive allocation of global and aliased variables and use of advanced compiler transformations mean that future architectures will need more registers to be efficient.

To summarize, the smart register file allows another class of variables to be allocated to registers. While this will allow better use of existing register sets, it will also tend to require more registers in the architecture to avoid spilling due to higher register pressure. Larger register sets are already appearing, as in the IA-64 architecture. This research will assume a register file larger than the typical 32 registers of a RISC architecture.

1.1.3 Eliminating load and store instructions

As a side-effect of having more registers and allowing aliased variables to appear in registers, SRFs reduce the number of load and store instructions necessary to move these variables between memory and the register file. This elimination of memory operations has several benefits. First, it reduces the amount of instruction fetch and decode that needs to be performed by the processor.

Second, it reduces the pressure on memory load and store ports, freeing them for more critical memory operations and allowing the hardware to exploit more parallelism. The trade-off is that the microarchitecture must be more complex so it can track aliased operations and maintain data consistency.

1.1.4 Prefetching to hide memory latency

The SRF that will be examined in this work allows the hardware to prefetch data when data addresses are placed into the register file. This is simple extension on existing instruction set architectures. Essentially, the load-effective-address instruction signals a prefetch of the data at that address, under the assumption that it will be referenced soon.

1.1.5 Scheduling loads above potential dependencies

In addition to prefetching, the SRF architecture allows load instructions to be moved above control flow and even data flow upon which the load is potentially dependent. It can do this safely, without generating spurious exceptions, because the data uses are left in their home blocks and exceptions are checked at the time of use rather than the time of preload. This also has the effect of prefetching but allows the prefetches to be scheduled farther from their use. This transformation, along with the others mentioned in this section, will be discussed in detail in later sections.

1.2 Foundational Assumptions

This section enumerates two assumptions that form the foundation of this work. First, we assume that binary compatibility is not of primary importance, unlike most previous work. Strict backward binary compatibility means that the architecture cannot be arbitrarily changed to accommodate new microarchitectural techniques. With the advent of binary-to-binary translation technology such as FX32! [55], this constraint can be removed so the compiler, architecture, and system designer is free to select a better point in the design space than previously allowed. Still, many of the techniques proposed herein can be applied directly to existing architectures with little modification.

Second, we assume that the compiler is capable of complex analyses and transformations. Work in the early- to mid-1980s assumed it was too expensive to do global register allocation because of compiler runtime or software bugs [17, 35]. We are not as concerned with compiler runtime as with the runtime of the generated code, especially in light of trends in processor speed and memory size that are evident in today's processors. We do not leave the job completely up to the compiler, though, as is the case in several VLIW architectures. We believe the best trade-off is somewhere in the middle, where strengths are taken from both the compiler and hardware. This philosophy may mean some duplication of effort as some things may be done by hardware that were also done in the software.

1.3 Outline

The remainder of this report is organized as follows. Section 2 describes previous research related to SRFs. Section 1 describes our first SRF designs and demonstrates the potential utility of it as well as relating it to some of the previous work.

2 Background and Related Work

This section on background work briefly outlines the various areas of research that are related to the smart register file (SRF). We start by discussing the basic trade-off between the use of registers and cache memory. We then look at basic register allocation and spilling and show how it fails to allocate variables to registers under conditions where

aliases are present. Alias analysis, a compilation step which is necessary for correct optimization, is examined next. The optimization called register promotion is then described; it uses the results of alias analysis and tries to alleviate the aliasing problem somewhat as it moves scalars from memory into registers in regions where the compiler is sure there are no potential aliasing relationships. We then briefly compare work in compiler-based control and data speculation to this research. The section ends with some summary remarks.

2.4 Registers, Caches and Memory

A fundamental trade-off in computer architecture is the structure of registers and cache memory in the processor. This trade-off will be examined in this section.

The benefits of registers are primarily short access time and short instruction encoding. Registers are accessed with direct addresses which simplifies value lookup. Since there are generally few register locations (compared to memory locations), the register address can be encoded in a few bits. However, registers complicate code generation because machine calling conventions typically require some registers to be saved across function call boundaries. This is an important consideration since function calls occur frequently [75]. There are many other trade-offs in the design of a register architecture for a processor. Table 1 catalogs them.

Cache memory structures have also been studied extensively, at least back to 1965 with the slave memories described by Wilkes [71]. Since our focus is on the register file, caches will not be considered here in any further detail. An excellent early survey is [72].

This research is concerned primarily with the problem that registers cannot generally contain aliased data. Even though registers have significant disadvantages compared to cache memory, it is desireable that aliased scalar variables (and even non-scalars) should be referred to through register names because of the sheer speed of register access. Caches and main memory can contain such aliased data because the hardware maintains consistency among the copies at the various levels. The movement of data from memory address space to the register address space has in the past meant that consistency could not be maintained simply because address information was not associated with the register data. Registers are meant for extremely fast access and adding hardware to keep this consistency must not be allowed to slow them down too much. This work is concerned with this very tradeoff.

The remainder of this section surveys some of the work related to the SRF. The keys to understanding all of this previous research is that it attempts do one or both of the following:

1. Reduce the number of memory operations
2. Reduce the apparent latency of memory operations

Both are essential to microprocessor performance because of the growing gap between processor and memory speed [56].

2.5 Register File Design

Research into the trade-offs between register files and caches has resulted in a wide variety of engineering solutions since the earliest days of computer architecture. Hardware

8

Registers	Cache
– storage size 1-128 registers (4B-512B)	+ storage size 256B-64KB typical
+ fast access (few and direct index)	– slower access (large, computed address, tags, memory management and protection checks necessary)
+ fewer address bits (less instruction bandwidth because of denser code)	– more address bits
+ lower memory traffic (fewer ld/st insts, cache and memory accesses, and power)	– more ld/st insts (expand the code again)
– aliases (computed addrs) and stale data	+ no synchronization
– more ld/st for synchronization at alias	+ no fcall boundary saves
– more ld/st at fcall boundary	+ data can be automatically kicked out at context switch boundary
– more ld/st at context switch boundary	– more costly to multi-port because of many entries
+ multiple ports less expensive because few entries	– hard dependence check (computed address)
+ easy dependence check for hazards	+ no aliases or stale data
– cannot take address of variable resident in (the C '&' operator)	+ can take address of variable resident in
– limited addressing modes (direct)	+ any addressing mode (computed)
– word-sized data only (ISA dependent)	+ any-sized data
– must have compiler to manage	+ dumb compiler will do

Table 1. Comparison of registers and cache.

registers (also called scratchpads) have been used since the early 1960s [38]. Table 2 gives a selected overview of this history. We will comment on a few of the machines listed in the table where they are relevant.

The CDC 6600 series machines had a total of 24 registers. Loads to 7 of the 8 address registers had interesting side effects. A load into register A[1,2,3,4,5] resulted in the data at that address being automatically loaded into data register X[1,2,3,4,5], respectively. Similarly, loading an address into A6 or A7 resulted in a store from X6 or X7 to that address. This allowed efficient encoding of vector operations because the load and store operations did not need to be explicitly specified. The Cray-1 later made this vector optimization explicit in its vector registers and instructions [64].

The Cray-1 [42] has a set of primary registers and a set of secondary or background registers. There are fewer primary registers, which allows them to be fast, while the secondary registers are slower but many in number. Long-lived values are stored in the secondary register files and promoted to the primary register files when used. The Cray-1 contains a total of 656 address and data registers (including the vector registers but not counting the control registers).

Date, Refs	Machine	Description
1961 [51]	Burroughs B5000	Stack-based computer. Registers hold the top two values on the stack. Eliminates some data movement introduced by stack.
1961 [41, 42]	Ferranti ATLAS	128 24-bit registers for data or address computation; 1 accumulator register
1962 [43]	ETL Mk-6	Still looking for info on this one...
1964 [42, 64]	CDC 6600	8 18-bit index regs, 8 18-bit address regs, and 8 60-bit floating point regs. Side effects of loading an address into an address register are described in the text.
1964 [77]	IBM System/360	16 32-bit integer regs, 16 64-bit floating point regs.
1966 [42]	TI Advanced Scientific Computer (ASC)	16 base regs, 16 arithmetic, 8 index, 8 vector-parameter regs, all 32-bits
1970 [42, 79]	PDP-11	8 16-bit integer regs (PC and SP included), 6 64-bit floating point regs. Extended to 16 integer regs in 1972.
1977 [42]	Cray-1	8 24-bit addr (A) regs, 64 24-bit addr-save (B) regs, 8 64-bit scalar (S) regs, 64 64-bit scalar-save regs (T), 8 vector (V) regs. A vector is 64 64-bits regs.
1978 [42, 80]	VAX	16 32-bit regs for integer or floating point. (PC, SP, FP, and AP regs included).
1978 [42]	Intel 80x86, IA-32	8 integer, 8-entry floating point stack (16-bits, extended to 32-bits later)
1987 [76]	Sparc	8 globals, 16-register window with 8 ins and 8 locals, as well as access to 8 outs which are the next window's ins, all 32-bits. Number of windows from 3 to 32. 32 64-bit floating point regs.
1987 [64]	AM29000	256 registers, all completely general purpose. 64 global, 128 "stack cache", 64 reserved.
1992 [81, 82]	Alpha AXP	32 integer, 32 floating point, 64-bits each
1998 [47]	IA-64	128 integer, 64 predicate, 128 floating point registers, some with rotating semantics for software pipelining.

Table 2. A partial history of hardware registers.

10

The hierarchical register file, which is very similar to the Cray-1 organization, is proposed in [73]. The authors present the classic argument that a large fast memory store can be simulated by a small fast store and a large slow store (in their case, 1024 registers). The results show speedups of 2X over a machine with only 8 registers. The trade-offs noted include higher instruction bandwidth and storage, larger context switch times, and increased compiler complexity. The instruction bandwidth and storage requirement is reduced by including an indirect access mode where a short specifier can be used to indicate the source value "comes from the instruction which is N instructions before the current instruction."

2.5.1 Large Architected Register Files

Sites presented perhaps the first in-depth discussion of the advantages of being able to support large numbers of registers, in his paper "How to use 1000 registers" [9]. He also noted the limitation caused by aliasing and coined the term *short-term memory* to denote a high-speed register set under compiler control. Besides cataloging some of the design issues related to short term memory systems, it was noted that it is often not possible to maintain data values in registers due to aliasing problems. If such values are placed in registers, they must be written and read from main memory as necessary to maintain coherence. Even though a machine may have 1000's of registers, it is likely that most of them will be left unused by conventional compilers (in 1979).

2.5.2 No Architected Register File

Work done in the 1980s at Bell Laboratories took a different approach by suggesting the complete removal of registers from the compiler-visible architecture. The work was embodied in the "C-machine" and its "stack cache", the "CRISP", and later the "Hobbit" [31, 32, 33, 34, 35]. Instead of programmer-visible registers, the architecture has a "stack cache" which is a special purpose data cache for program stack locations; as such, it caches references to local scalar, array, and structure variables in the function linkage stack. The goal of this cache was to eliminate register allocation from the compiler and reduce the amount of data movement at function call boundaries. This allows the use of a large number of hardware registers (entries in the stack cache) without needing compiler allocation and without requiring every implementation of the ISA to have the same number of registers. Initial proposals were for 1024 registers in this cache, but the first reported implementation had 64 entries [34].

The essential features of the stack cache that distinguish it from a normal data cache are as follows: 1) it has no tags; 2) it caches a contiguous range of memory, i.e. the top of the program stack; and 3) the range being cached is delimited by high and low address registers. Alias checking is enabled by comparing any computed address with the high and low ranges of the stack cache; if the address falls within the limits of the high and low bounds, the data is in the stack cache. If not, memory is accessed (there is no other internal data cache in the processor). In this way, data objects can be allocated to the stack cache without fear of aliasing (because the stack cache is in memory address space). Special handling is needed when the stack cache is overflowed, but this is rarely the case. The

authors found that a large percentage of data addresses can be computed early in the pipe-line because they are simple base+offset calculations where the base is the stack (or frame) pointer. The stack pointer remains constant for the life of the function (except when call-ing out to children functions).

The stack cache was designed to incorporate the best features of both registers and cache memory. It was direct mapped and had no tag comparison, so it was fast. The instruction encoding only required a short stack offset from the current stack frame, much like the short direct register specifier of a conventional architecture. The stack cache could hold strings, structures, and other odd-sized data. Finally, the compiler could take the address of a variable in the stack cache.

It is important to note that the C-machine research assumes that compilation is expensive and that compilers are hard to write correctly. Therefore, simplifying the com-piler was the motivation for the decision to eliminate register allocation in favor of the more straightforward stack allocation of local variables. The single-pass compilers in the 1980s were not able to determine if a variable could be placed in a register because of aliasing. The requirement of simple compilers is no longer widely held, as evidenced by the large number of optimizing compilers for register-based architectures. In fact, later versions of the CRISP compiler used an optimization similar to register allocation to pack variables into the stack space in order to reduce stack cache misses. The other primary assumption in the CRISP work is that function calls are frequent and that overhead of the function linkage mechanism is very important to overall performance. This is still true today ([74, 75]) so an important criterion of a register-architecture is how it handles func-tion calls.

2.5.3 Register Windows

The Sparc architecture's register windows [76] are a hybrid register/memory archi-tecture intended to optimize function calls. It is a cross between the C-machine's stack cache and a conventional single-level register file. Each subroutine gets a new window of registers, with some overlap between adjacent register windows for the passing of function arguments. Because the windowed register file is large and many ports are required to implement parallel instruction dispatch, Sun researchers proposed the register cache and scoreboard [48, 49]. The register cache takes advantage of the locality of register reference and the fact that register file bandwidth is not utilized efficiently for large multiported files. This is another fundamental trade-off between registers and memory. Sun and others report that about 50% of data values are provided by the bypass network [48, 49, 50] and there is an average of less than one read and 3/4 writes per instruction. The Sun work also noticed that a small number of the architected registers are heavily used (stack and frame pointer, outgoing arguments, etc.). Because of these factors, the register file cache can be quite small and still capture a large portion of the register references. Fully associative reg-ister caches of size 20 to 32 were found to have miss rates of less than a few percent. This can provide a significant savings in cycle time and power consumption compared to the 140-register file in the SPARC architecture (for an implementation with 8 windows [76]). Other architectures that could benefit from a register cache include the IA-64 and AM29000 because they have a large number of architected registers. It is unclear from the previous work what the compiler could do to more evenly utilize the register file.

There is other work that attempts to reduce the implementation cost of large register files. One is a technique called "virtual-physical registers" which is described in [45]. Here the goal is to allocate physical registers as late as possible so that their live ranges (in terms of processor cycles) are reduced. This is based on the observation that the actual lifetime of a value begins at the end of instruction execution rather than when the instruction is decoded. The difference could be a large number of cycles. Tags, called virtual-physical registers, are used to specify instruction dependencies, but these have no storage associated with them. The actual physical register storage is not allocated until instruction writeback. This has the effect of either 1) increasing the perceived instruction window size or 2) allowing the window to be reduced in size without negatively affecting performance. The second option is interesting because it allows the processor to implement a smaller number of physical registers. The only difficulty is that sometimes the processor may run out of physical registers and the instruction cannot be written back. In this case, the instruction is re-executed.

2.6 Register Allocation and Spilling

The problem of allocating scalar variables to registers, called the *register allocation problem*, is usually reduced to a graph coloring problem [12, 13, 14, 85], where an optimal solution is well-known to be NP-complete. Other research has cast the problem as set of constraints passed to an integer programming solver [22, 23], or bin packing [86]. We focus on graph coloring in this work because it is the most common technique for optimizing compilers. This section outlines some previous work in expanding the register set of an architecture so that the compiler can do more effective allocation and spilling.

Mahlke et. al. examined the trade-off between architected register file size and multiple instruction issue per cycle [4]. They found that aggressive optimizations such as loop unrolling, and induction variable expansion are effective for machines with large, moderate, and even small register files, but that for small register files, the benefits are limited because of the excessive spill code introduced. Additional instruction issue slots can ameliorate this by effectively hiding spill code. This work noticed little speedup or reduction in memory traffic for register files larger than about 24 allocatable registers (often fewer registers were required). We hypothesize that because of a conventional application binary interface [88] and traditional alias management the compiler was not able to take advantage of any more registers.

Register Connection is an approach used by the IMPACT research group which adds registers to the architecture. It does so in a way that is very careful to maintain backward compatibility and requires a minimum of changes to the instruction set architecture. Connect instructions map the logical register set onto a larger set of physical registers instead of actually moving data between the logical and physical registers. This is similar to register renaming [83, 84] but is under compiler control so that register allocation and code optimization and scheduling can take advantage of the larger set of registers available. This technique is helpful for instruction sets with very few registers (8-16) but does not help much after 32 registers (where not much spill code is generated). The connection instructions were carefully designed to minimize execution delay and code size.

The compiler-controlled memory [8] combines hardware and software modifications to attempt to reduce the cost of spill code. The hardware mechanism proposed is a small compiler-controlled memory (CCM) that is used as a secondary register file for spill code. The compiler allocates spill locations in the CCM either by a post-pass allocator that runs after a standard graph-coloring allocator, or by an integrated allocator that runs with the spill code insertion part of the Chaitin-Briggs register allocator. A number of routines in SPEC95, SPEC89, and various numerical algorithms were found to require significant spill code, but rarely were more than 250 additional storage locations required to house the spilled variables. Potential performance improvements were on the order of 10-15% but did not include effects from larger traditional caches, write buffers, victim caches, or prefetching. These results show the potential benefit of providing a large number of architected registers–not only simplifying the compilation process in the common case, but also reducing spill code and memory traffic.

2.7 Alias analysis

Compiler alias analysis is yet another field related to the SRF. Alias analysis is important because it enables optimizations such as common sub-expression elimination, loop-invariant code motion, instruction scheduling and register allocation to be applied correctly to the program. While alias analysis is used to determine potential data dependencies for all of these optimizations, we view it as taking two distinct roles. The first is in register allocation, where it determines whether a variable can be *allocated* to a register or not. The second is in code transformation, where it determines whether a code *transformation* is legal. While both kinds of decisions are necessary for correctness (the overriding concern), the first is a data layout decision and the second is a code-layout decision. Alias analysis is used to ensure correctness of an optimization but if it is conservative it limits the scope and potential of applied optimizations. In other words, alias analysis is necessary, but *aggressive* alias analysis is needed to allow good optimization.

In deciding how the code-layout can be changed, the compiler is deciding whether it is semantically correct to move code out of loops, to eliminate redundant computations, or to otherwise re-arrange the code.

When the compiler addresses the data-layout problem, it must trade off the speed of the allocated memory against the functionality of it. In the case of an on-chip, direct addressed register file, the speed is very high but its functionality is low because data is accessed by statically specified indexes. Furthermore, the conventional register file does not have built-in checking for aliases between data in a register and data in memory.

The remainder of this section is organized into subsections describing the various previous research. These could also be divided into software, hardware, and combined hardware/software solutions.

2.7.1 Background on Alias Analysis

A location in a computer's memory is referred to by a numerical address which is computed during the execution of any instruction that accesses that particular location. Memory aliasing occurs when a storage location is referenced by two or more names. This

can happen in languages like C that have pointers. Data at a memory location can be temporarily kept in a register only if we can assure that all instructions that might refer to that memory location can be made to refer to the register instead. Because instructions compute the address of the data they refer to at their time of execution, it is often impossible to tell before execution (i.e. at compile time) which instructions refer to a particular memory location; thus we run the danger of substituting two or more registers for what appears to be different memory locations, when we should have substituted only a single register. If this occurs, copies of the same data will be placed in two or more registers, leaving open the possibility that the copies can be changed separately. Thus data that was meant to represent the value of a unique variable can end up with two or more distinct values. Clearly this is wrong.

The allocation of data to registers is done by a compiler–the program that translates a programming language like C into basic machine instructions. The compiler analyzes a program before it executes and thus cannot detect if address aliasing does occur when the program runs. To avoid possible errors the compiler must make conservative assumptions about the values of addresses, and, as a consequence, must be conservative about what data can be kept in registers. This in turn means that whole classes of data cannot be placed in registers, at least for part of their lifetime.

Aliasing through memory is problematic because modification of a value through the use of one name will change the value examined through another name when both names refer to the same location in memory, (e.g., a[i] and a[j] may refer to the same location). If the compiler can determine with certainty that the names refer to disjoint locations, it is possible to allocate each name to a machine register where the value will reside. Similarly, if the compiler can be certain that both names always refer to the same location, it is possible to replace uses of both names with a single register name and allocate the location to a machine register.

Unfortunately, making such determinations is difficult. The use of pointers or accessing of arrays with different index variables creates new names. Furthermore, the pointer or index can be modified programmatically, thus changing the names at runtime. Such locations cannot be easily placed in registers because a traditional machine register has only one name.

Such values can be allocated to registers within regions of the program where the compiler can determine the exact set of names that refer to the location. In fact, this is a necessity in a load-store architecture, because the memory value must be placed in a register before use. However, such allocations are short-lived, because a modification of the memory value through another name will not be reflected by a change in the value in the register. Thus, the value must be updated by re-loading the value from memory. Likewise, any modification of the value through the register name must be written out to memory in case the value is accessed through an alias.

Languages with stronger typing than C allow the compiler to make more assumptions during alias analysis because only those names which have the same type as the variable can refer to the variable.

2.7.2 Modern Processor Memory Disambiguation

The conventional disambiguation hardware in a microprocessor (see for example [10]) is not open to the compiler. This forces the compiler into the very conservative mode described in the last section, which requires loads and stores around the references to aliased data. Furthermore, loads cannot be moved past branches or stores on which they may (or may not) depend.

2.7.3 Static Analysis

The fact that aliasing information is not provided by the hardware to the compiled code forces most compilers to do static alias analysis to prove correctness of optimizations. Examples can be found in [16, 18, 28, 29, 87]. These references represent a range of complexity in the analysis phase; compile time is an important consideration in such analyses because they are so complex.

As far as register allocation is concerned, the simplest approach is to note which variables are potentially aliased and then simply not allocate them to registers. For code motion, simple heuristics can be employed to determine whether a load has a potential dependence on a previous store.

When the aliasing relationship between two instructions is not known, they can be moved relative to each other conditionally by the software. This is done by runtime memory disambiguation [65], where explicit comparison instructions are used to route the code to the best execution path. If two addresses do not match, then the better code schedule can be selected. In the case they do match, the original, less aggressive code schedule must be provided.

2.7.4 Register Promotion

In function-level[1] register allocation, a variable is typically marked as 'allocatable' or 'not allocatable' depending on whether it can be resident in a register for its entire lifetime or not. In a conventional compiler and processor, a variable cannot be placed permanently into a register if there is more than one name used to access that variable. For the C language, if the address of the variable is taken the variable is said to be aliased and cannot be placed in a register. Global variables are also typically marked 'unallocatable' because register allocation algorithms are designed to run at the function level instead of the program level. Those variables which cannot be permanently allocated to registers are left in memory and require a load before each use and a store after each definition.

Register promotion [27, 24, 25, 26] allows aliased variables to be placed into registers in code ranges where aliasing cannot occur. The variable is *promoted* to a register by loading it from memory at the beginning of the range. At the end of the range, the variable is *demoted* back to memory so that subsequent definitions and uses through other names are correctly maintained. It is apparent that this optimization increases register pressure

---

1. "Function level" register allocation is typically called "global" register allocation. We use the former to avoid overloading the term "global."

C Code	Without Register Promotion	With Register Promotion

```// i potentially```
 ```// aliased somewhere```
 ```// but not in loop```
 ```while (ctr) {```
 ```   i = i + 1;```
 ```   ctr--;```
 ```}```
	``` beq rctr,0,L2```
	 ```L1: ; loop body```
	 ``` ld ri, -8(sp)```
	 ``` add ri,ri,1```
	 ``` st ri, -8(sp)```
	 ``` sub rctr,rctr,1```
	 ``` jmp L1```
	 ```L2: ...```
		``` ld ri, -8(sp)```
		 ``` beq rcond,0,L2```
		 ```L1: ; loop body```
		 ``` add ri,ri,1```
		
		 ``` sub rctr,rctr,1```
		 ``` jmp L1```
		 ```L2: st ri, -8(sp)```

Figure 2. A example of register promotion. Without register promotion, the load and store are executed every time around the loop whereas with promotion, the loop has no loads or stores.

because more values are maintained in registers during the non-aliased regions. The loads and stores are removed from these regions.

Several variants have been examined which use different code regions as the basic unit of promotion. [24] considered loops as the basic range for promotion; [25] used arbitrary program intervals as the promotion regions; and [26] did not consider explicit program regions but instead used a variant of partial redundancy elimination to remove unnecessary loads and stores. All of the previous work shows substantial reductions in the number of dynamic load instructions executed and varying reduction in the number of stores eliminated.

The promotion loads and demotion stores can be placed in infrequently executed paths in the control flow graph; this is shown to require more static loads and stores but results in fewer dynamic loads and stores. Register pressure was shown to increase by up to 25% in the most common cases [25].
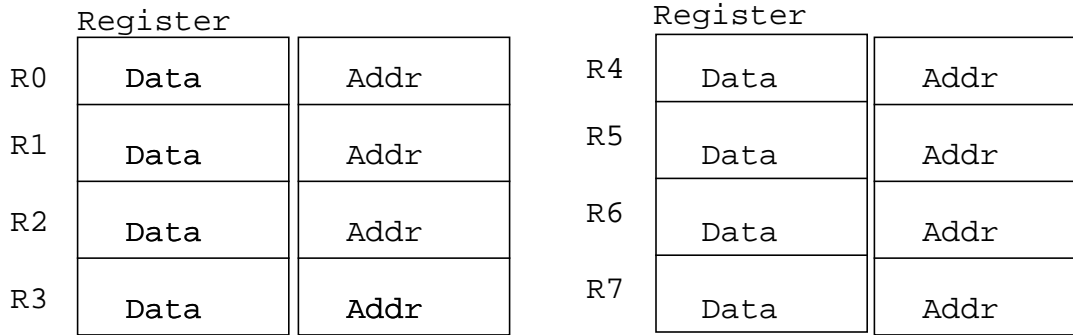
The studies typically show that 0-25% of loads are removed and up to 40% of stores are removed, depending on the program.

Explicit load/store instructions are needed for register promotion, and the compiler must demote a value to memory whenever there is a potential aliasing relationship.

2.7.5 CRegs

The Short-Term Memories described by Sites are the inspiration for CRegs [5, 6, 7]. CRegs solves the aliasing problem much the same way as the Smart Register File proposed here. Registers have associated address tags which are checked against loads and stores to keep the registers and memory consistent. However, because the compiler may assign aliased variables to different registers, memory contents potentially have duplicates in the CReg set. On a store, an associative lookup must update all copies of the data in the CReg array.

The associative update feature requires that the maximum number of aliased items be limited to a small number so that the update hardware does not become too cumbersome. The register file is divided into sets (similar to a set-associative cache) and the compiler must assign aliased data items into a given set. The modifications to the compiler for analysis and register assignment were discussed in [5]. In practice, there were found to be at most two names for a datum, indicating that the extent of associativity only needs to be

Register					Register			
R0	Data		Addr		R4	Data		Addr
R1	Data		Addr		R5	Data		Addr
R2	Data		Addr		R6	Data		Addr
R3	Data		Addr		R7	Data		Addr

Figure 3. A diagram of the partitioned CRegs register file.

about 2 or 4 to satisfy most programs. In the case where there are more potential aliases for a datum, the datum is simply allocated to memory and only loaded when used and immediately stored upon definition. A diagram of a partitioned register file is shown in Figure 3.

CRegs reduces the number of memory operations by eliminating redundant loads and stores from the program. These loads and stores were introduced in the conventional architecture because of aliasing. Simulations in [7] indicate CRegs can eliminate up to 23% of memory operations for an overall decrease of up to 14% of total dynamic instructions. No CPI-based performance numbers were given, but these instruction counts indicate that CRegs (or similar techniques such as the SRF) can provide significant performance benefits.

A CRegs design based on the MIPS-X processor core is proposed in [6]. The design shows possible techniques to implement data forwarding in the presence of aliases. The 32-register file is divided into 8 sets of 4 registers each. Notable is the way that alias information is computed and stored in "mask" bits with each register. This allows aliases to be recognized early in the pipeline. The mask bits then control the forwarding and update of data in the processor pipeline.

Figure 4 shows an example adapted from the earliest CRegs paper. The conventional code in the second column shows that the store to a[i] requires the compiler to reload the values of a[j] and a[k] because it is uncertain of the runtime values of i, j, and k (note the boldface **ld** instructions). If i were to be the same as j, then the update of a[i] also updates a[j]. The goal of CRegs is to eliminate these extra load instructions. The third column shows the CRegs code that implements the array calculations. The reloads have been eliminated. The store is there to update memory as well as forward the new value to the registers for a[j] and a[k] *if there is an alias*. Since a[k] already has an address associated with it, the hardware knows where to put the new value of a[k] and can lazily write it back.

2.7.6 Variable Forwarding

Variable Forwarding, or register forwarding as it is is alternately known, was proposed as an improvement to CRegs [90]. It attempted to achieve the same benefits as CRegs, namely reduction in alias-synchronizing memory traffic while improving on CRegs in two primary ways: 1) eliminate the need for compiler alias analysis before regis-

18

C Code	Conventional Assembly	CRegs Processor
| ```
void foo(int i,j,k)
{
  a[i] = a[j] * a[k]



  a[k] = a[j] + a[k]
}
``` | ```
ld raj,0(ra,rj,4)
ld rak,0(ra,rk,4)
mul rt,raj,rak
st 0(ra,ri,4),rt
; reload a[j],a[k]
ld raj,0(ra,rj,4)
ld rak,0(ra,rk,4)
add rt,raj,rak
st 0(ra,rk,4),rt
``` | ```
ld raj,0(ra,rj,4)
ld rak,0(ra,rk,4)
mul rt,raj,rak
st 0(ra,ri,4),rt


add rak,raj,rak
``` |

Figure 4. Code demonstrating the CRegs ISA. The destination appears on the left. The notation 'ra' indicates the register which contains the base address of array a. 'rai' signifies the register which contains either the value of a[i]. The target machine supports base+scaled index+offset address mode, like the x86. The boldface load instructions are needed to synchronize raj and rak with the valued stored to memory. The dashed arrows in (a) and (c) represent the potential dependency that exists between the store to a[i] and the second reads of a[j] and a[k]. The solid arrows in (b) show the load instructions that must be inserted by the conventional compiler to handle the case when their is actually an alias from the store. The code in (c) shows that CRegs can eliminate two loads and the final store instruction.

ter allocation; and 2) simplify the implementation. Load instructions map the load address with the register where the value is loaded. If another register is already mapped to the address, the original register and the new register are combined into a *forwarding group*. The original register is the *leader* of the forwarding group and accesses to the data mapped to the leader, whether through memory or through another register in its forwarding group, is forwarded to the leading register. When the leading register is stored (thus unmapping it), it is removed from the group and the other members of the group must be told their new lead register. This update requires a doubly-linked list of pointers amongst the registers in the forwarding group so that each member can be visited and updated with the proper leader information. An associative memory maintains a mapping from memory address to register leader (if any). No performance results were presented.

The elimination of compiler alias analysis simplifies the software side of the problem but complicates the hardware over CRegs because a value can be mapped to any registers in the register file.

2.7.7 IMPACT and EPIC

Work done at Illinois on the Impact EPIC architecture [46] is concerned with scheduling load instructions ahead of control dependencies and aliased stores. Allowing loads to move past stores in the instruction schedule has a large impact on performance because otherwise the scheduling is very constrained. In previous work, the same researchers proposed the memory conflict buffer (MCB), which associates addresses with registers and tracks later writes to the addresses [63]. In this way, a load can be scheduled above a store and the hardware will report when an aliasing condition actually occurs. Explicit check instructions which access the MCB state are required to determine if an aliased memory operation occurred between the check and the earlier, hoisted load

instruction (called a *checked load*), at which time recovery code can be initiated. The recovery code and check instructions increase both the static and dynamic instruction counts, but the speedups reported are significant for benchmarks limited by memory ambiguities. The later research [46] is concerned with generating efficient recovery code.

If a checked load instruction is followed by instructions which use the loaded value speculatively, exception information can be propagated through the uses so that one check can happen at the end of a long string of code. This reduces the number of check operations.

The Merced implementation of IA-64 utilizes a hardware structure very similar to the memory conflict buffer call the Advanced Load Address Table (ALAT). It allows the IA-64 compiler to advance load instructions and associated uses beyond branches and stores [47]. To propagate exception information through a string of instructions, a NaT bit is employed. There is one per architected register. When the NaT bit is set on a register (say because of a page fault), all subsequent instructions which use that register essentially become NOPs and set their output register's NaT bit.

2.7.8 Memory Renaming

Tyson and Austin proposed memory renaming which allow loads to execute early in out-of-order processors [10]. This optimization is done entirely in hardware with no modification to the binary. This is achieved by tracking the loads and stores that frequently communicate with each other. Once a stable relationship has developed between a load and a store, the load's data can be accurately predicted to be coming from the store it is associated with. This allows memory to be bypassed entirely in the critical path–the address and the data are both predicted at once by the producer-consumer relationship between the load and store. A value file contains the data shared between the load and store. The key to early (speculative) resolution of the load is that the load and store PC's are used as the index of the value in the value file.

The prediction must be checked by performing the actual load, but this is off the critical execution path. The authors found that some of the load-store pairing comes from aliased data and global data, which they assume cannot be allocated to registers.

The producer-consumer relationship between stores and loads is highly predictable (about 70%) and allows their memory renaming technique to achieve an average 14% speedup across the SPEC integer benchmarks. The stability between a store and its consuming load indicates that aliasing does not happen very often in practice.

2.7.9 Other

Other work has focused on early generation of load addresses, prediction of load addresses, or prediction of load values in order to speed up program execution. None of these techniques assumed compiler involvement and thus worked with conventional binary programs.

2.8 Summary

This section has outlined a number of areas of research that are related to Smart Short Term Memories. Because the SRF relies on both hardware and software support, the previous work is a large body of computer architecture research. The previous work can be divided into three major categories: 1) that which deals with register file design; 2) that which deals with register allocation; and 3) that which deals with memory alias analysis and optimization in the face of aliasing. All of these are important foundational work to the SRF designs considered in the rest of this paper.

3 Preliminary Studies

This section details work that has been done to lay the foundation for the dissertation. First we examine one architecture for a "Smart Register File," or SRF for short. Then we present some initial experimental results. These show that the SRF potentially could have a significant impact on performance because about 10% of local variables and even higher percentages of global variables cannot be allocated to registers for their full lifetime because of aliasing. Static and dynamic analyses are shown to verify this result.

3.9 The Smart Register File

The goal of the Smart Register File (SRF) is to allow aliased data to be specified through a short register name during the region where aliasing occurs. As we have discussed in the previous section, the register promotion optimization can be used to allocate aliased values to registers in regions where they are not aliased. But this optimization still requires loads and stores to surround such regions so that the value resides in memory during the regions of aliasing. In the region where the value is aliased, it must still be accessed by explicit load and store instructions.

The SRF is conceptually a register file separate from the general purpose register file. Each SRF register contains an address. When the SRF register is specified as an instruction operand, the hardware does a load from the address contained in the SRF entry to get the data (on a data read) or a store to write the result (on a data write). Therefore, the SRF is a register file with an implicit level of indirection which is executed transparently by the hardware. Many architectures have special-purpose address registers and many CISC architectures can access data indirectly like the SRF. The Motorola 68000 and Intel IA32 architectures are examples [60, 61] where operands can be specified indirectly. The CDC 6600 even automatically loaded or stored a data register based on a change in the contents of an address register.

By putting the addresses of aliased data items into the SRF (instead of the data themselves), it is possible for the hardware to keep those data items in a data cache and index it by the SRF contents. The data cache is part of the memory hierarchy. In this way, we are attempting to utilize the benefits of both registers and caches. The SRF and data cache together form a Smart Short-Term Memory because the compiler manages allocation of addresses into the SRF and the hardware takes care of data value maintenance in

the data cache. In effect, the compiler is sharing alias information with the hardware to increase overall performance.

The cache can be managed as a traditional cache where the hardware stores data back to a lower level in the memory system only if it is dirty. The hardware does memory dependence checking in the same way it does in a conventional microprocessor design [67, 68].

The remainder of this section discusses in more detail how the SRF functions. In particular, it explains how the SRF may help processor performance in the presence of aliased variables.

3.9.1 Smart Register File Implementation

The SRF can be implemented in two ways. The first way is a register file which is separate from the general purpose register file. The most significant bit of the register specifier can be used to distinguish which register file is being requested. An 8-bit register specifier can therefore provide access to 128 general purpose registers and 128 indirect registers. Because of this split design, special instructions are required to move data between the SRF and the GPRF or to source data from both register files in a single instruction. This is because in the most common programming languages, data and addresses are interchangeable, such that a data value produced as the result of an arithmetic operation is used later as the base address of some memory operation. The instruction opcode would have to specify which register file the sources come from and which register file the destination goes to. Data that are later used as addresses would have to be moved from the GPRF to the SRF.

The second (and preferred) implementation is as part of the existing general purpose register file. The most significant bit in the register specifier in this case indicates which indirection level the hardware should use to access the data (0=data is in the register, 1=data is in memory at the address specified in the register). Thus an 8-bit register specifier in this scheme provides access to 128 registers, any of which can be used in normal mode or indirect mode depending on the setting of the most significant bit in the register specifier. This unified configuration does not require special instructions to move data between registers, as SRF entries and GPRF entries are mixed in a single register file; any register can be used in either direct or indirect mode at any time.

Throughout the rest of this section, we will assume a configuration like the second one. A diagram is shown in Figure 5. R2 is being used indirectly as indicated by the 1 bit in the indirection field of the instruction. We say that R2 is being used as a "smart register." For the add instruction depicted, the second source's value is in R3 but the first source's value is accessed indirectly through R2. The result of adding 5 + R3 is placed directly into register R1.

3.9.2 An Example

This example is taken from the CRegs work [5]. Figure 6(a) shows a small C function which demonstrates the problem with aliasing. At compile-time it is not known whether i equals j or k. In a conventional architecture, a[j] and [k] need to be reloaded

```
      R2: Addr = 100

         Register File
```

```
  Address      Value

    100          5
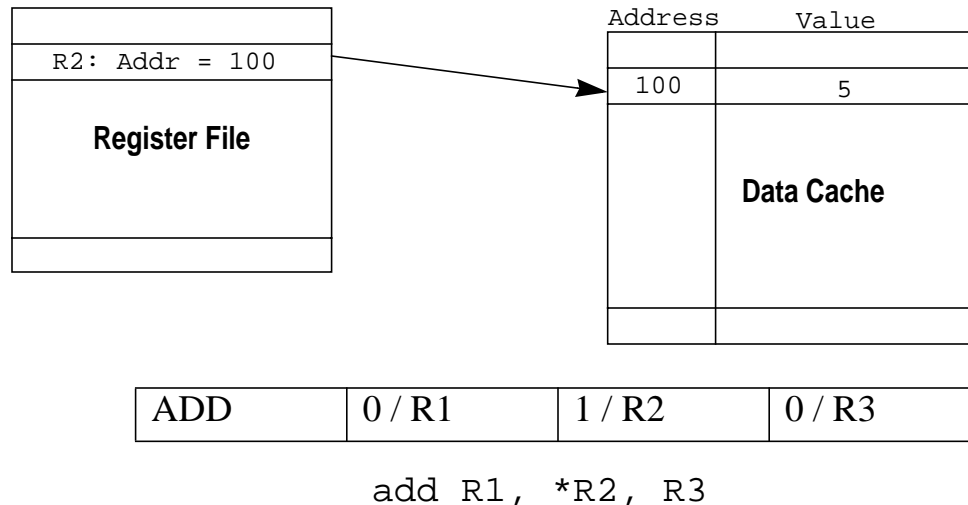
         Data Cache
```

| ADD | 0 / R1 | 1 / R2 | 0 / R3 |

```
add R1, *R2, R3
```

Figure 5. A diagram of the smart register file, cache, and instruction format. The add instruction shown is formatted as a RISC instruction, with an additional bit per register specifier. This indirection bit tells whether the register is used directly (0) or indirectly (1).

after the store of a[i] because the store to a[i] may have changed the value of either a[j] or a[k] (or both if i equals j equals k). These extra load instructions are shown in boldface type in Figure 6(b). The compiler does not know at compile time the values of i, j, or k, so it must include these load instructions.

Notice that there are three fewer instructions in the SRF sequence (Figure 6(c)). First, there are no explicit store instructions. The second load of a[j] and of a[k] have been eliminated. These instructions have been replaced with simple **map** instructions which load the address of the data into their respective indirect registers. As a side effect, the data at the address is prefetched in anticipation of a future use. The multiply and add operations can access their values indirectly and no loads and stores are explicitly specified.

If i equals k, the hardware will conceptually forward the store of a[i] to later uses of the data at that address, no matter which register is used to specify it. The hardware may actually do forwarding directly or through the data cache.

Since the **map** operation operates on addresses, it can be scheduled in advance of the uses of the values. The prefetch side effect is quiet in that exceptions are not reported until the value is used indirectly through the register (this would be a program error).

3.9.3 Smart Register File Design Considerations

The SRF has some of the benefits of both the register file and the cache (see Table 1). Namely, it allows instructions to be encoded using small register specifiers, even though these specifiers map to long addresses through the SRF entry. It eliminates the need for loads and stores whose purpose is to synchronize data between the register file and memory address spaces, thus reducing the number of instructions fetched and executed.

But when we say "elimination of loads," what do we mean? An operand specified with a '*' tells the hardware that it can find the data in the data cache. While this is

C Code	Conventional Assembly	SRF Processor
| ```
void foo(int i,j,k)
{
  a[i] = a[j] * a[k];




  a[k] = a[j] + a[k];
}
``` | ```
ld raj,0(ra,rj,4)
ld rak,0(ra,rk,4)
mul rt,raj,rak
st 0(ra,ri,4),rt
; reload a[j], a[k]
ld raj,0(ra,rj,4)
ld rak,0(ra,rk,4)
add rt,raj,rak
st 0(ra,rk,4),rt
``` | ```
map raj, 0(ra,rj,4)
map rak, 0(ra,rk,4)
map rai, 0(ra,ri,4)
mul *rai,*raj,*rak


add *rak,*raj,*rak
``` |
| (a) | (b) | (c) |

Figure 6. Code demonstrating the SRF ISA. Here, 'rai' signifies the register which contains either the value of a[i] (in the conventional assembly column) or its address (in the SRF column). Columns (a) and (b) are as in Figure 4. The code in (c) shows that the SRF can eliminate these two extra load instructions.

implicit and not specified in the instruction stream, it is still a load operation from the perspective of the hardware.
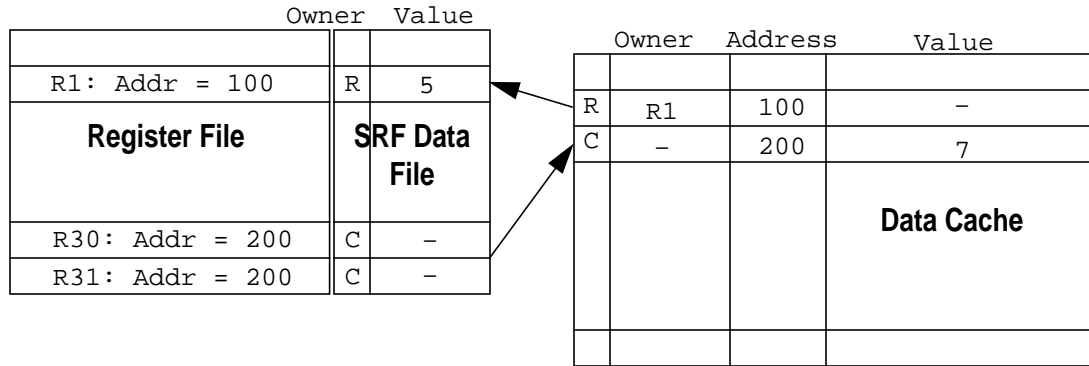
So it appears that we have not gained any benefit except shrinking the size of the instruction stream. It should be noted, however, that this is a significant benefit. Fewer instructions mean fewer cache misses [89]. Register allocation and traditional optimizations such as loop-invariant code motion speed up program execution by eliminating loads/stores and moving code out of loops. Fewer instructions also means fewer fetch and decode cycles which are very expensive on some processors.

Finally, some may suggest that there are a lot of free execution cycles in the typical superscalar processor, i.e. that "instructions are free." We argue the reverse. First, instructions are only free if they are largely independent of the algorithm (not on the critical path). Second, the more complex semantics of a SRF instruction provide a well-defined interface which the hardware can implement in a variety of ways depending on speed and cost considerations. For example, the hardware may decide to decode the more complicated indirect instructions into RISC-like micro-operations much like the Pentium III core does, immediately recovering the instruction stream as if it had the loads and stores in it. Or it may decide to do direct execution on the indirect instructions.

3.9.4 SRF with Data File

A more clever implementation can keep the value with the SRF entry instead of in the cache, and eliminate even the implicit load from the cache. This is illustrated in Figure 7 where R1 is the "owner" of address 100 and thus the value can be maintained with the register file in a structure we call the SRF data file. This condition is indicated by the "register/cache ownership field" (R/C) being set to R to indicate register ownership. Both the cache and the data file have a copy of this field. For the cache, the R/C field serves as a valid bit for the owner register number. If R is set, then the register owner field is also set. If C is set, then the cache owns the value and the register owner field is not applicable. The

	Owner	Value
R1: Addr = 100	R	5
Register File	**SRF Data File**	
R30: Addr = 200	C	–
R31: Addr = 200	C	–

Owner	Address	Value
R	R1	100	–
C	–	200	7
		Data Cache

Figure 7. Eliminating load and store operations with the SRF data file.

SRF data file uses the R/C field to determine when the value field is valid. If R is set, then the value is valid because the SRF owns the address of that value. If C is set, then the value must be obtained from the cache (or memory hierarchy).

Further stores through *R1 (notice the star) can write their value directly into the data file. This is feasible because a given data value is not aliased in certain regions of the program (recall register promotion). The cache does not need to be accessed as long as R1 "owns" the memory location. The data written can be trickled back to the cache at the hardware's convenience or at serialization points.

If there is an alias for the data item, then the cache becomes the owner for it and the processor reverts back to implicit loads and stores on every access. Figure 7 shows a case where both R30 and R31 contain address 200. The data is owned by and stored in the cache and the R/C field is set to C in the ownership fields. Stores through *R30 and *R31 are routed to the cache because the cache owns the data.

The only remaining problem is how to maintain the ownership relationship. An aliasing condition occurs when one SRF entry contains an address and another one is set to the same address. The data value that was associated with the first entry needs to be moved back to the cache so it can be "shared" with the new SRF entry. When an address is written into an SRF entry, the cache is queried to see whether another entry has the same address. If not, then the data value can be associated with the SRF entry directly, and the cache is told that the register file owns the data (R) and which register is the owner. Otherwise, the original register owner is divested of its ownership of the value and both SRF entries must access the data indirectly through the cache. In this case, the data at the address is owned by the cache and is marked C in both cache and SRF data file.

One shortcoming of the SRF with data file mechanism is that once an aliasing condition is encountered, the cache always owns the value. We are currently investigating how to handle this. Another problem is how to handle register ownership in the case that the cache datum owned is evicted from the cache because of a miss to a conflicting datum. Owned lines could be locked down in the cache so that this would not happen. Or, they could be un-owned at the time of eviction; the next access would cause a cache miss and re-own the datum.

This design avoids the associative searches which have been present in previous designs. Whether this is implementable with reasonable cost and whether it is necessary to speed up the operation of the SRF-based processor are matters for experimentation.

The assumption is that aliasing actually happens infrequently. The compiler is allowed to omit the loads and stores in the common, unaliased case. The hardware manages the data in the infrequent case that an aliasing condition does occur. An optimization to this approach is to have the compiler indicate the frequency of aliasing between two variables and let the hardware decide if and when to enregister the value. If aliasing is frequent for a certain address, the hardware could decide to leave the value in the cache all the time.

Optimizing to keep an aliased value in the register file has demonstrated a problem: when we mix the semantics of memory and registers too closely, it is difficult to keep values consistent in both address spaces. This is precisely the problem that earlier CRegs work attempted to overcome [5, 6, 7]. In that work, the compiler was required to allocate aliased items to a subset of the registers. The hardware had to do an asociative write-update whenever a value was written into the register file. The associative search in CRegs is unwieldy and probably not practically implementable, particularly in a multi-ported register file at clock rates acceptable to the market. By the ownership relationship for the data at a particular address, we can replace the associative search with a direct lookup in the data cache tag store. Only in the case when this tag indicates that there is an aliasing condition do we need to inform the owning SRF entry to "un-own" the value and submit it to data cache management (this requires a data file read and a cache write). This can be done directly without an associative search because the cache already knows which register owns the value–it can go to the register directly and inform it of the aliasing condition.
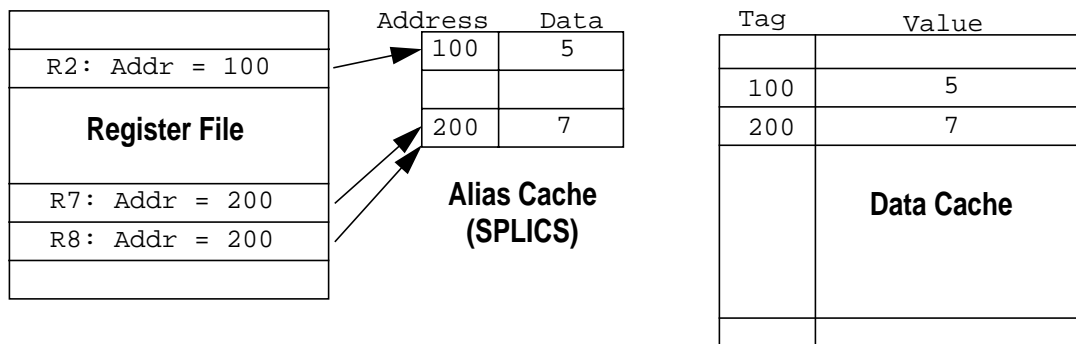
Finally we consider the problem of SRF entries that point to bytes and halfwords and overlapping data. The instruction which initializes the SRF entry (say, a **map**) can also mark the register as pointing to byte-sized data. Then when the hardware uses the SRF entry indirectly in a later instruction, it can format the incoming (loaded) or outgoing (stored) data properly. Two extra state bits are required at each register to implement this (00=byte, 01=16-bit, 10=32-bit, 11=64-bit). The extra bits at each register do not pose a great problem. The Intel IA64 architecture already provides for a 65th bit on each register that specifies whether an exception has occurred during the production of the value in the register (the bit is called the Not-a-Thing bit or NaT bit [47]). The tag store in the cache may need valid bits for each byte of data. Overlapping data structures such as C unions are a problem for this mechanism and are thus far handled as in the conventional compiler/ architecture. We leave these things for future work.

3.9.5 SRF with Alias Cache

The final SRF implementation we consider in this report is based on the Alias Cache. The alias cache is diagrammed in Figure 8.

The alias cache is designed to overcome some of the problems with the SRF Data File. Specifically, we note that the data file implementation had two problems. First, management of the ownership relation is cumbersome. Second, the ownership information stored in the cache is very sparse and thus wasteful of storage. The alias cache remedies these two problems by compressing the sparse ownership information in the data cache into a small side cache which is accessed in parallel with the L1 data cache. The alias cache has several advantages:

1. It is small, so it will return data sooner than the data cache.

Figure 8. The SRF Alias Cache.

2. It is indexed by memory address, so it is in the memory space and can be managed by the hardware like a cache as far as dirty writeback, LRU allocation, etc. are concerned.

The data file access time is very fast since the data is often owned by the register; we trade some of this speed for easier bookkeeping. For these reasons, we feel the alias cache has a more promising future than the SRF data file.

Various aspects of the alias cache will be examined as part of the ongoing studies in this research project. Parameters of interest include its size, associativity, and number of ports.

The alias cache implementation of the SRF can be viewed as a cross between the SRF ISA and the Variable Forwarding associative memory, except in our case the associative memory (alias cache) contains the actual data instead of pointing to a register that contains it. Since the alias cache is the only repository for aliased data, the implementation is not required to implement a doubly-linked list in order to keep track of the single location for the data. Instead, the data is kept in one place and all registers refer to it indirectly.

3.9.6 Smart Register File Benefits

We have already mentioned that the SRF implicit indirection level is similar to the addressing modes in CISC architectures. CISC ISAs allow memory operands to be used without explicit load and store instructions. The SRF proposes to add this access semantic back to a RISC architecture, in essence to close the gap between direct-only access to registers and the completely flexible addressing of the first level of the memory hierarchy. This has several potential benefits, some of which this section outlines.

3.9.6.1 Elimination of Loads and Stores

The SRF allows the compiler to eliminate load and store instructions necessary for alias synchronization; these operations are required in a conventional RISC instructions set architecture. In a conventional ISA, whenever the compiler is not sure whether a data value can be modified by a store, it must insert a load after the store to update the value in the register. When the SRF is used, the aliased value can be used through the SRF. The hardware has to "load" the data from the data cache, but the load does not need to be spec-

ified in the instruction stream–it is implicit. This is shown in the example given in Figure 1. As we discussed above, the SRF with data file can even eliminate the implicit loads and stores.

3.9.6.2 Prefetch

A load into an SRF entry (a "load effective address" operation) effectively signals a load (or prefetch) of the data at that address. If the data is already present in cache and some other entry in the SRF contains the same address, then we have avoided a data load in the presence of an alias. Data which the compiler deems important can be prefetched simply by issuing a load effective address operation early in the instruction stream.

3.9.6.3 Miscellaneous Benefits

Members of complex data types such as structures or arrays can be kept in SRF registers. Structures and arrays are often aliased because of pointers to them or indexes into them. Therefore the conventional compiler cannot easily put critical data fields of records or arrays into registers. The SRF allows these values to be kept in registers.

3.9.6.4 Hardware Management

The hardware can decide when to execute store operations because it can track when data is dirty. If a data item is not modified, no store operation is necessary. Thus a data value in the data cache that is aliased with a later load need not be stored to memory before the load occurs. We call this the elimination of dirty stores. This was also demonstrated in Figure 1 by the elimation of the two store instructions.

3.9.6.5 Reduction in Memory and Register Port Pressure

The SRF ISA allows a reduction in use of register and memory ports. As shown in Figure 9, if the variable x is resident in memory, then incrementing it can be implemented more cheaply by the SRF processor than the conventional RISC because no temporary register is required. If we ignore the **map** operation required to load the address of x (suppose that this instruction is amortized over many increments of x), then the SRF processor saves two register writes. If the data file and ownership status bits are added to the SRF, the implicit load and store operations can also be eliminated.[1]

The SRF ISA may also allow a reduction in the number of address computations because addresses of data are stored in the SRF entries. This is similar to the common-subexpression elimination optimization when applied to addresses. As shown in the example of Figure 1, the conventional processor executes six address computations. The SRF executes three and stores them in registers for future use. This frees integer units for work more relevant to the algorithm. While CSE can eliminate address computations, the conventional processor requires registers to store the addresses as well as registers to store the

---

1. Though somewhat contrived, this example serves to illustrate the point. Obviously, a good compiler will allocate the local x to a register. We could make x a global or a structure field instead.

C Code	Conventional Assembly	SRF Processor
```/* * local x in memory */ int x; x = x + 1;```	```/* 1 mem read, 1 mem write 2 reg writes, 2 reg reads, */ ld rt, [x] add rt, rt, 1 st [x], rt```	```/* 1 mem read (maybe), 1 mem write (maybe), 1 reg write, 2 reg reads */ map rx, [x] add *rx, *rx, 1```
(a)	(b)	(c)

Figure 9. SRF reduction of register and memory port pressure. The SRF code contains one less register write than the conventional code.

values when they are loaded from memory. The SRF only requires one compiler-visible register for both.

3.9.6.6 Speculative Code Motion

The SRF allows loads to be statically scheduled above branches upon which they may depend. The indirect register may be loaded with a NULL pointer, for instance, but the hardware will not signal the exception until the instruction which uses the register indirectly. This is called control speculation in the literature [46, 47].

An example is shown in Figure 10. The load inside the if statement has been scheduled above the comparison and changed to a **map**. All uses of val inside the if are also translated to uses through an indirect register. Since the uses only appear inside the home block of the load, we guarantee that no exceptions are generated that would not have been generated by the original assembly. A quiet load (deferred exception) or prefetch [76] would have a similar result except that two instructions would be specified–the prefetch and another load at the home location.

In the SRF architecture, no explicit check instructions are required since the hardware can carry the exception information with the register value and only signal the exception when the value is used. For example, if the hoisted SRF load is to an illegal, protected, or unmapped page in memory, the exception is deferred until the use of the data from the SRF register.

Furthermore, the **map** instruction need not have a special opcode indicating a speculative mode. Checks are not required because no fixup is required in the case of an exception. (If an exception happens, it happens at the point of use, which has not been moved from the original code; in this case, the exception was a program exception and the program will halt.)

With a check instruction, as in EPIC [47], uses of the loaded value can also be moved above the branch and the check can be placed in the home block of the load and uses so that only one check is needed; in this case, the hardware simply forwards the exception information through the data dependencies. Section 2 describes this in more detail.

C Code	Conventional Assembly	SRF Processor

`if (ptr != NULL) {`
`  val = ptr->data;`
`    ... use val ...`
`}`
	`cmp rptr, 0x0`
	`beq L1`
	`ld rval, +16(rptr)`
	`    ... use rval ...`
	`L1:...`
		`map rval, +16(rptr)`
		`cmp rptr, 0x0`
		`beq L1`
		`    ... use rval ...`
		`L1:...`

(a)	(b)	(c)

Figure 10. An example showing how the SRF allows control speculation.

With the appropriate hardware and software, this can be extended to allow data speculation. Data speculation allows the scheduling of loads above potentially dependent stores. Check instructions are also required for data speculation because an incorrect speculation requires fixup of some sort. The hardware verifies that this was legal, and if not, the software provides a fixup routine. The fixup routine can either be out-of-line code which the processor branches to [63] or inline with the assistance of predication to enable only certain portions to execute in the fixup mode [46].

3.9.6.7 Summary of Benefits

The advantages of the SRF fall into three categories. First, the SRF allows the removal of load and store instructions which reduce the number of instructions executed by the processor. The second advantage is that it allows more registers to be used. Register allocation is an important optimization which is often limited in scope because of aliasing among variables; the SRF allows aliased variables to be placed in registers. Third, the SRF allows some speculative motion of loads above branches and aliased store instructions. The effect is a prefetch operation, which reduces perceived memory latency.

3.9.7 Smart Register File Design Space

The SRF instruction set architecture presented above allows any register to be used in either direct mode, as in a conventional processor, or indirect mode. The compiler makes the decision at compile time which registers are directly specified and which are not. It may use a single register in both ways in certain types of code. Obviously if no registers are used in indirect mode, the machine is reduced to a conventional architecture. When reduced in this fashion, we can reclaim the indirection bit to double the number of specifiable registers.

The other end of the spectrum is to only allow indirect access through the registers. Such a processor would have only SRFs. Here again we could reclaim the indirection bit in the register specifier to double the number of registers that can be specified. This architecture requires that we perform an "address allocation" step in the compiler to compact the addresses into the number of available registers. This is analogous to the register allocation step in current compilers. We do not pursue this design because the SRF processor presented allows both traditional and indirect accesses to registers. The compiler can determine where it falls along the spectrum (from all direct registers to all indirect regis-

30

ters) based on profiling or other information. This is, of course, at the cost of the indirection bit in the register specifier.

3.10 Applying the SRF to Existing Instruction Set Architectures

Because of the extra indirection specification necessary for the registers, the SRF architecture is not trivially compatible with existing RISC instruction set architectures. However, there are are least two options that still allow the proposed technique to be used in commercial computers:

1. Partial recoding of an existing ISA. The indirection specification can be made in some instructions but not allowed in others. For example, there may be available bits in the three-operand instruction encoding (in the function field, for example).
2. Full recoding of an existing ISA. The ISA could be recoded to a larger instruction format (say 48 or 64 bits) and binary translation techniques could be used to translate from the old ISA to the new. Binary translation is becoming a well-established technique that allows designers to break old design constraints while allowing old code to run correctly [55].

Note that many CISC architectures already have the SRF-like semantics. The Intel IA32 architecture can specify memory operands as part of most operations. The most basic way to do this is indirectly through a base register. This is specified in assembly with parentheses around the value, as in (eax). Our SRF notation would say *eax. The SRF presented here only allows this base-register indirect address mode instead of the full complement of address modes (base+scaled index+offset) allowed in the IA32 ISA.
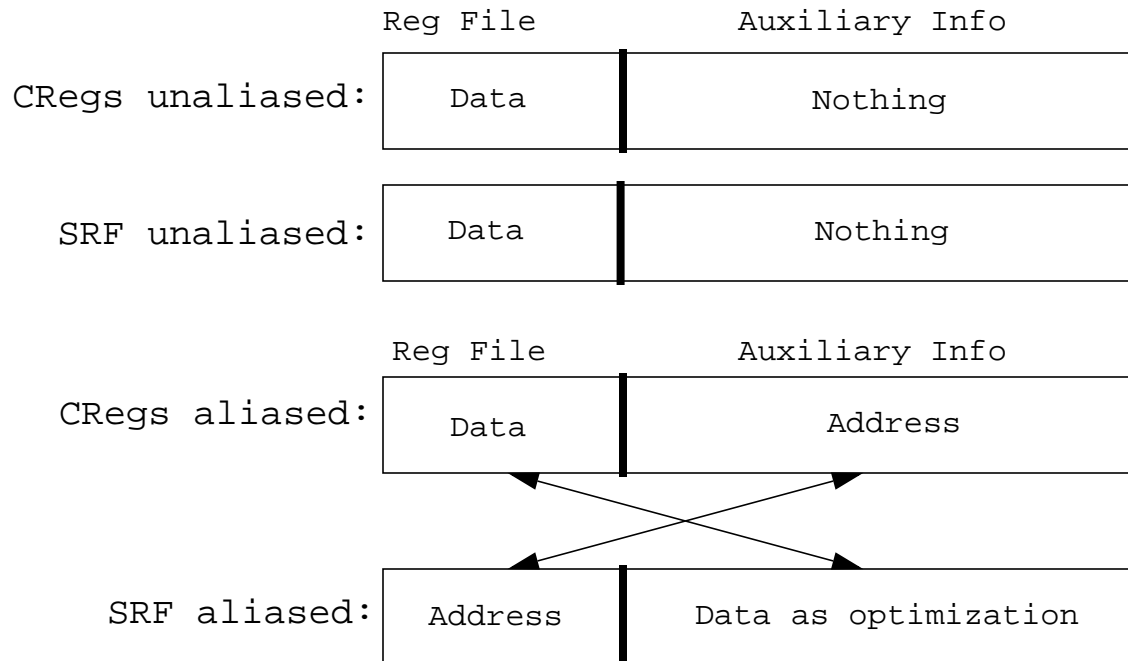
3.11 Comparison with Previous Work

The SRF is similar to a lot of the previous work which we have discussed in the previous section. The most notable difference is the number of different ideas that it attempts to bring together. For example, CRegs removes load and store instructions and IA64 allows loads to be rescheduled above stores. The SRF allows both.

Memory renaming [10] attempts to connect a load with the store producing its data value in order to short-circuit the path through the memory system. That work showed that memory renaming is particularly effective for aliased data as well as global and heap data. The SRF allows the compiler to put more of these kinds of data into registers and completely eliminates the load and store instructions necessary in the conventional architecture. The stability found in the renaming work implies that aliases happen infrequently.

The SRF can work without register promotion if a simple compiler is desired, or in concert with promotion if there are large ranges where it would be beneficial to use a scalar through a normal, direct register.

Figure 11 diagrams the basic difference between CRegs and the SRF. For data which is not aliased, both the SRF and CRegs store the data in the register and need no auxiliary mapping information for bookkeeping. When data is aliased, though, CRegs keeps the data in the original register file entry and *augments that with address information*. The SRF puts the address in the original register file entry and *augments that with the*

```
              Reg File            Auxiliary Info
           ┌──────────────┬┬──────────────────────────┐
CRegs unaliased: │   Data      ││          Nothing          │
           └──────────────┴┴──────────────────────────┘

           ┌──────────────┬┬──────────────────────────┐
 SRF unaliased:  │   Data      ││          Nothing          │
           └──────────────┴┴──────────────────────────┘

              Reg File            Auxiliary Info
           ┌──────────────┬┬──────────────────────────┐
CRegs aliased:   │   Data      ││          Address          │
           └──────────────┴┴──────────────────────────┘

           ┌──────────────┬┬──────────────────────────┐
 SRF aliased:    │  Address    ││    Data as optimization   │
           └──────────────┴┴──────────────────────────┘
```

Figure 11. Contrasting the SRF and CRegs.

data as a performance optimization. The SRF maintains only one copy of the data, however, even if the same address resides in two or more registers. The important difference is that SRF keys its memory accesses by the *address*. The address is read-only; read-only items do not have an aliasing problem, so it does not matter how many copies of the address are present into different registers. Furthermore, the address is a level of indirection away from the data. This allows the hardware to maintain a single copy of aliased data, but with multiple addresses pointing to it. On the other hand, CRegs uses the *data* as the key; since data is written, it can be aliased, and copies of the data can reside in multiple registers. When a write occurs CRegs must synchronize all copies. SRF does not have this problem.

3.12 Initial Experiments

The first experiments are designed to determine the potential of the SRF design. All experiments are run on the SPEC95 integer benchmarks [69] with the MIRV compiler on an IA32 processor. Recall that the IA32 has 8 general purpose registers, 6 of which are available for user variables.

3.12.1 Register Pressure

The first question is to determine how many variables are actually allocated to registers and what is the register pressure in each function. The data in the second column of Table 3 shows the maximum number of live local variables in the SPEC95 integer bench-

marks. The data was gathered immediately before register allocation in the MIRV back-end. The number cited is the largest number of simultaneously live integer scalar values in any function in the benchmark; most other functions in the benchmarks have much lower register pressure. The conclusion is that 10 to 20 registers is enough for local variables. This agrees with several previous papers in the literature. Note that no compiler optimizations are turned on except for global copy propagation, constant folding, and strength reduction. With optimizations such as loop unrolling, function inlining, and software pipelining these numbers would be higher.

The third column of Table 3 cites the number of local variables that were not allocatable because of aliasing (in the functions which had the highest register pressure). They are marked unallocatable for their entire lifetime because of the potential for aliasing. These variables require synchronizing loads and stores whenever they are referenced, or at least loads and stores for promotion/demotion. It is interesting to note that in addition to the maximum live local variables, there is often another 10% of variables that are not allocatable to registers for their entire lifetime. The SRF can handle these variables and eliminate all the loads and stores associated with them.

The vortex benchmark is notable because it has a function (C_MapRefToDb) which has 35 variables live at one location in the program and 8 additional variables that are not allocatable. It has 49 local variables and 10 incoming parameters. It passes some of its local variables by address to other functions. This is a common cause of aliasing.

Benchmark	Maximum Live Locals	Unallocatable Local Variables
compress	11	1
gcc	41	5
go	21	2
ijpeg	31	0
li	9	5
m88ksim	15	0
perl	20	0
vortex	35	8

Table 3. Maximum live local variables and unallocatable local variables.

3.12.2 Global Variables

We see that there is not much need for more than 32 registers given the traditional way of doing register allocation. However, for these benchmarks, there are a significant number of global variables, as shown in Table 4. Accesses to these globals are responsible for a large number of loads and stores in the programs, and thus we conclude that allocating globals to registers is very important. Allocation of global variables is the first way to make use of a large number of processor registers. We can envision utilizing a register file of 128 entries where 50 or more registers are devoted to global variables. We do not yet

have lifetime range data for these global variables but future experiments will determine if this is a worthwhile idea. Work by Wall [70] indicates that allocating globals to registers is an important optimization.

Benchmark	Global Variables	Allocatable	Not Allocatable	% Not Allocatable
compress	29	28	1	3%
gcc	1019	923	96	9%
go	83	71	12	14%
ijpeg	32	31	1	3%
li	79	79	0	0%
m88ksim	107	105	2	2%
perl	211	205	6	3%
vortex	604	389	215	36%

Table 4. Global variables statistics in SPECint95.

The last column in Table 4 shows the percentage of global variables that cannot be placed into a register for their entire lifetime. This percentage is computed by linking together the benchmark at the MIRV IR level and processing the MIRV IR with a filter that determines whether the global variable's address is ever taken. If it is, then we say the variable cannot go into a register (even though it may be able to be enregistered for parts of its lifetime). The integer benchmarks fall into two categories. For the majority of the benchmarks, most global variables are allocatable for their entire lifetime. Two interesting cases are *go* and *vortex*, where there is a high percentage of globals that are aliased. In both benchmarks, most of the variables in question are passed by address to some function which changes the global's value. The number of call sites where this happens is usually fairly small for any given variable. In vortex, a heavily used variable called *Theory* is modified through a pointer in memory management code. It appears that this use through the pointer is for initialization only (so register promotion could promote the global to a register). The SRF allows these globals to be enregistered, providing even further benefit from a large register file.

The data shown above does not quantify the importance of the variables examined. It is possible that the globals are not used very frequently. The aliased globals may be even less important, in which case the SRF would not provide much performance benefit. In order to determine if this is the case, we have run experiments to determine the frequency of use of global variables in the program.

The results of the first experiment is shown in the 16 graphs of Figure 12 at the end of this section. It shows histograms for static usage counts of global variables in a program. The static usage count of a variable is an estimate of dynamic usage frequency determined by the following formula:

$$\text{static frequency count} = \sum_{\text{all uses and definitions}} 2^{\text{loop nest level}} \qquad (Eq.\ 1)$$

34

That is, a reference to a variable within a function is given weight 1 (2^0). A use within a loop is given weight $2^{\text{loop nest level}}$. This is a simplified version of a standard compiler formula for computing a weighted usage count without having to profile the program [16]. It is used to indicate a variable's relative importance for the sake of register allocation or other optimizations. The formula is using the heuristic that a variable used in a loop will probably be used by several iterations of the loop and is more important than a variable not used in a loop. This estimate is neither a lower nor an upper bound on the usage frequency of the variable because uses may be reached more or less frequently depending on the control flow that the program actually follows.

The static frequency count histograms show the static frequency count information for each of the SPECINT95 benchmarks. For each benchmark, there are two graphs. The one on the left shows the frequency count information for the global variables that are allocatable to registers. The graph on the right shows the frequency count information for the global variables that cannot go into registers. For example, for compress95 (Figure 12 upper left), there are 6 global variables which have a static frequency count of 2 or less. 4 globals have a frequency count of 3 or 4. 1 variable has a frequency count between 100 and 1000. The graph for compress95's unallocatable variables shows that 1 unallocatable global variable has a count of 9 or 10. The sum of the heights of the bars in both graphs for compresss95 is equal to the number of global scalar variables in the benchmark.

Since the data presented in Figure 12 is only static, we cannot draw broad conclusions from it. However, it does show that for most benchmarks, the static frequency counts do not rise above 30 to 40. The notable exception is go, where there are 17 globals which have frequency counts above 100. Four additional globals which are heavily used are not allocated to registers. This indicates the importance of both allocating globals to registers and also allocating aliased globals to registers. In the case of go, the globals are global configuration parameters which are assigned and compared against in many places in the benchmark.

The second experiment is analogous except the benchmark is executed and global variables annotated with a dynamic frequency count, so the data is much more meaningful. Counters are used to keep track of the execution frequency of the block statements in the program; these are then used to compute how many times a given global variable is accessed. This data is more accurate than the static frequency count estimate. The graphs are shown in Figure 13. The bins are chosen to be the same as in the static frequency count case, even though some variables are used more than 1 million times.

The dynamic frequency graphs for globals show that most of the time only a few percent of the unallocatable variables are actually used very much (compress95, ijpeg, li95, m88ksim and perl). The notable exceptions are go and vortex, where the frequently used unallocatable global variables are a significant proportion of all the frequently used globals (roughly 20% and 33%, respectively). We now compare these dynamic usage counts with the static data in Table 4. For go recall that 12 out of 71 (14%) of global variables are not allocatable. The dynamic frequency count information shows that about 40 of those variables are "heavily used,", i.e. in the leftmost bin in Figure 13. There are an additional 10 unallocatable variables that are also heavily used. Thus, about 10 out of 50 important variables (20%) are aliased and not able to be placed into registers for their entire lifetime. Similarly for vortex, 45 out of 145 variables are heavily used, or about

33%, compared to the 36% static estimate. The static percentage is giving a good approximation to the importance of aliased and unaliased variables in the program.

These high dynamic usage counts indicate that for go and vortex, performance could be improved (perhaps significantly) if these globals were allocated to SRF registers, which do not require load and store operations.

It is also interesting to compare the static and dynamic data and see if they are in agreement. The dynamic frequency graphs in Figure 13 show that the dynamic usage is somewhat bimodal, at least for the bin configuration that we have selected. Many variables are either used very little or are used many times. The static graphs in Figure 12 do not show this quality. Instead, variables are bunched toward the left side of the graphs in many cases, indicating that the typical global variable is only used in a handful (a couple of dozen or less) locations in the program source code.

This section has provided information about the global variables in the SPECint95 programs and shown that allocation of global variables to registers could have a very significant impact on performance because there are so many globals. For certain classes of benchmarks, global variables are aliased a lot and could benefit from the SRF design presented earlier in the section. The next section does a similar analysis for local variables.

3.12.3 Local Variables

Table 5 shows how many local variables appear in the static representation of the SPEC programs and of those how many can be allocated to registers for their entire lifetime. The data show that for go, li, m88ksim, and vortex, a significant percentage of variables have their address taken and cannot be allocated to a register for their entire lifetime. The SRF processor could allocate these variables to indirect registers. Of course, some of those variables could be placed into registers for part of their lifetime using the register promotion optimization mentioned earlier. The following are the reasons why so many local variables are used by address:

1. In go, there are many calls to list manipulation functions which take a pointer to an index into the list and potentially modify it
2. In li, the first parameter (a pointer to a list node) of many functions is passed to another function called xlarg. It is passed by address and modified.
3. In m88ksim, there are many calls to a function rdwr() which reads or writes the simulator memory; it takes a pointer to a value which is either read from (on a memory write) or written to (on a memory read). Error messages and statistics printing are also a major source of addresses being taken.
4. In vortex, there are many call sites to TmFetchCoreDB where a pointer to a pointer is passed; the pointer is modified.

Of the remaining benchmarks (compress, gcc, ijpeg, and perl), only 2 to 4% of variables are not allocatable to registers. It should be noted, however, that these small percentages of variables are responsible for a high percentage of the scalar loads and stores in the program. They are still important to optimize.

The data in the Table 5 does not show what happens when aliased data is included in the allocation because we do not yet have a simulator capable of executing code for the SRF-based processor. It also does not consider local or global complex data types like structures and arrays, whose individual elements can be allocated to SRF entries.

Benchmark	Local Variables	Allocatable	Not Allocatable	% Not Allocatable
compress	116	114	2	2%
gcc	19811	1905	754	4%
go	3608	3503	105	3%
ijpeg	3175	3155	20	1%
li	1806	1516	290	16%
m88ksim	1658	1562	96	6%
perl	2512	2476	36	1%
vortex	11970	10660	1310	11%

Table 5. Local variable statistics in SPECint95.

Static and dynamic frequency counts for all local variables in SPECINT95 are shown in Figure 14 and Figure 15, respectively. Similar conclusions can be drawn for local variables that were drawn for global variables. The benchmarks go and vortex show a significant number of heavily used local variables whose addresses are taken; these are potentially aliased. These two are joined by li95. Li had no aliased global variables, but there are at least 40 aliased variables which are used heavily compared with the 300 or so unaliased variables.

The dynamic frequency graphs show the same kind of bimodal distribution that was noted earlier regarding the frequency counts of global variables. This information can be used to guide register allocation decisions to keep the most important variables in registers.

3.13 Summary

This section has introduced the smart register file and discussed several potential advantages to this structure over current microprocessor architecture. These benefits come mainly from eliminating explicit load and store operations as well as allowing prefetching and code scheduling flexibility. The SRF allows more classes of values to be allocated to registers.

The second half of the section shows some experimental evidence that there are significant amounts of data in the aliased and global classes which the SRF attempts to enregister. The benchmarks which seem to be most amenable to SRF optimization are gcc, go, vortex, and li. These benchmarks generally have high percentages (more than 10%) of variables that cannot be allocated to registers using conventional means. Future work will quantify the actual performance improvement that is possible.
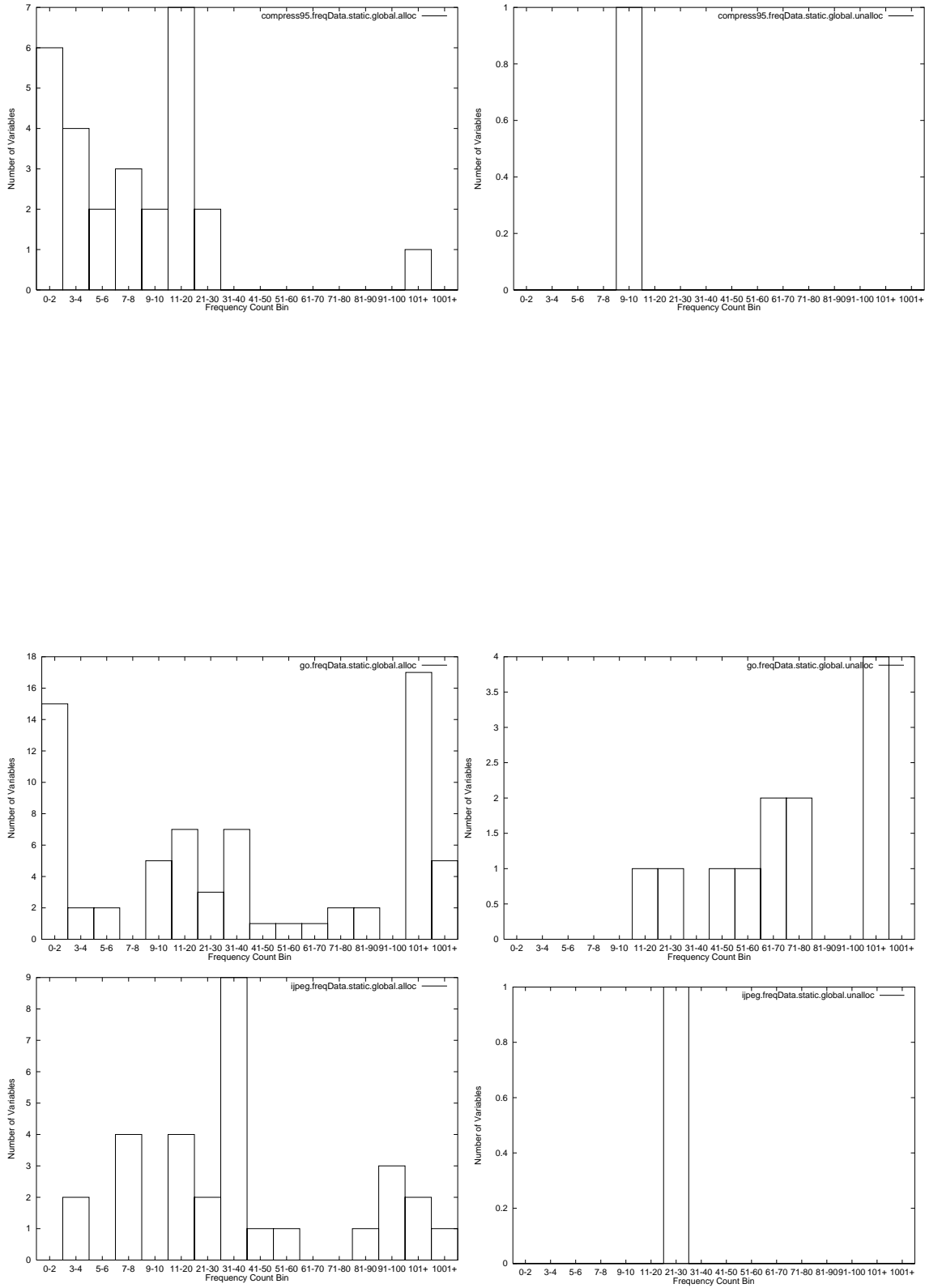
Acknowledgments

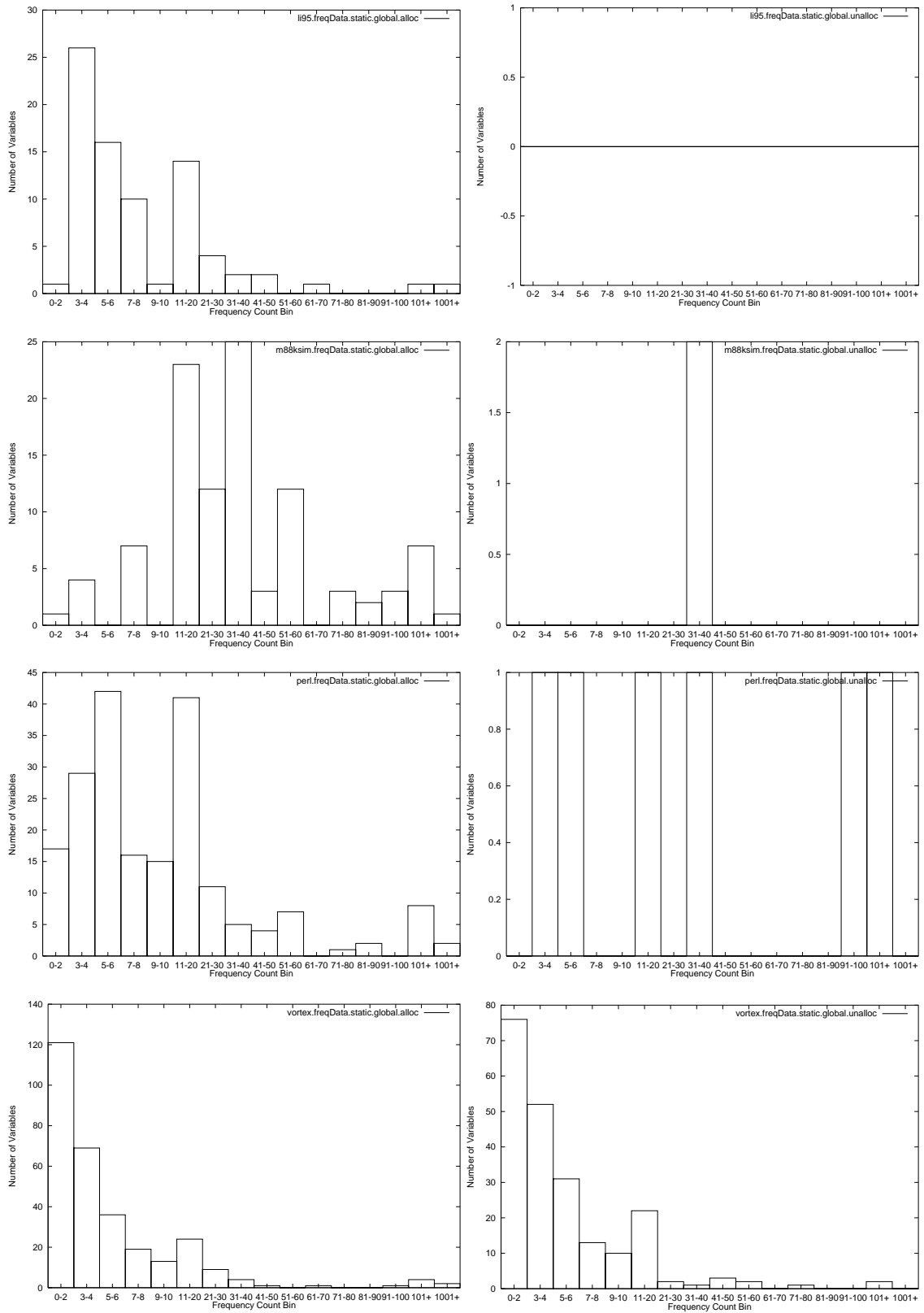**Figure 12. Static global variable frequency counts for SPECint95.**

Figure 12 (cont). Static global variable frequency counts for SPECint95.

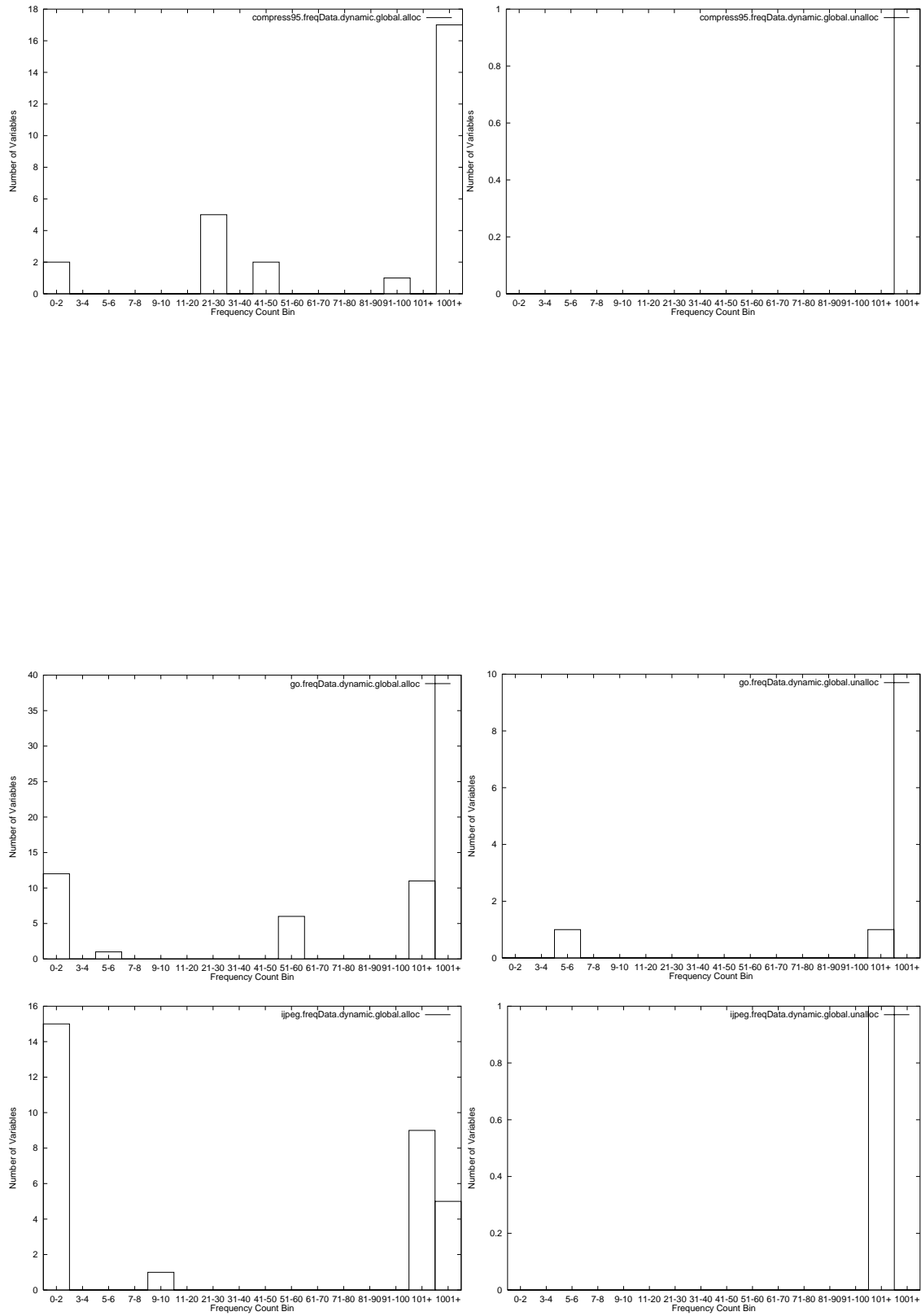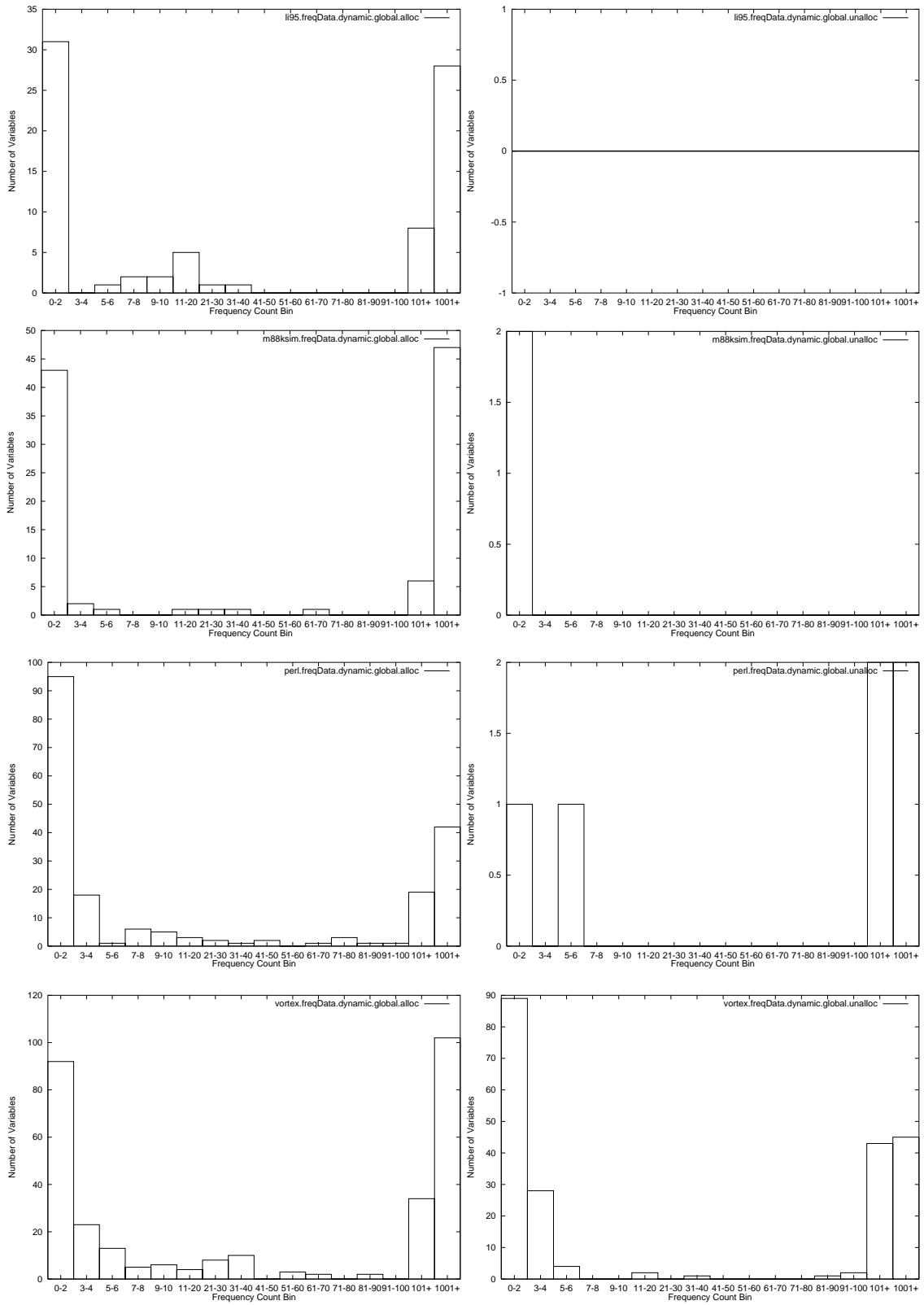**Figure 13. Dynamic global variable frequency counts for SPECint95.**

Figure 13 (cont). Dynamic global variable frequency counts for SPECint95.

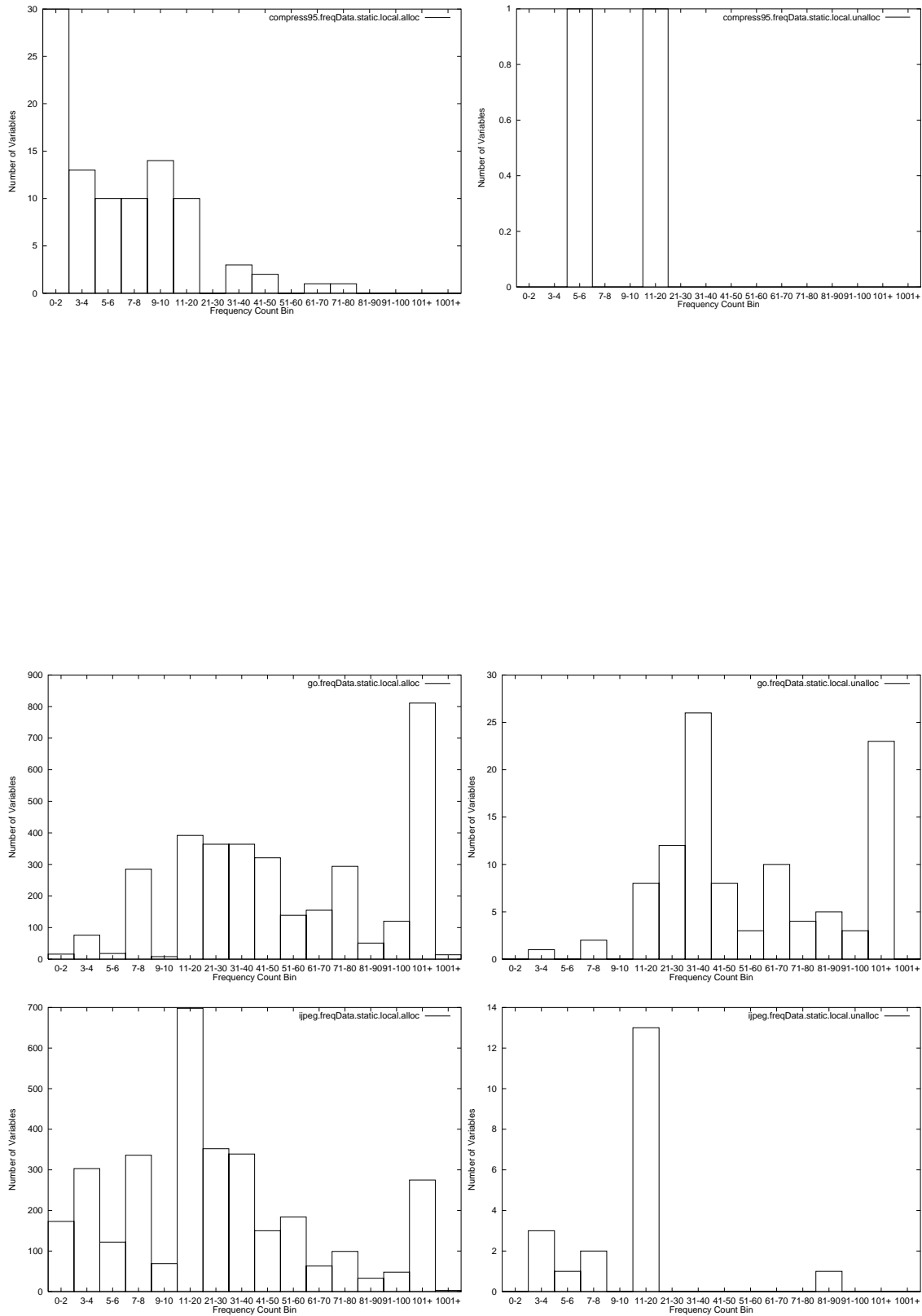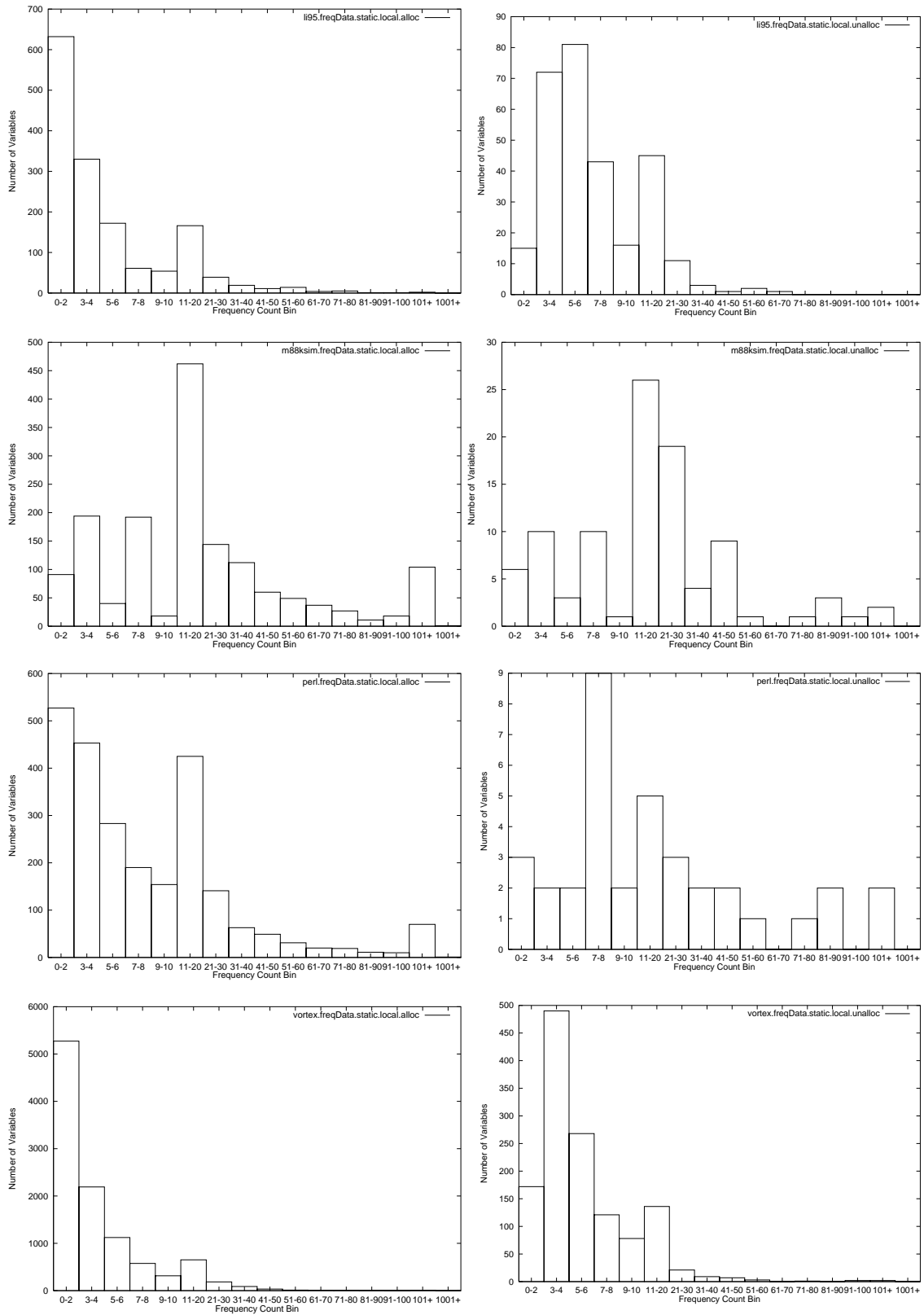**Figure 14. Static local variable frequency counts for SPECint95.**

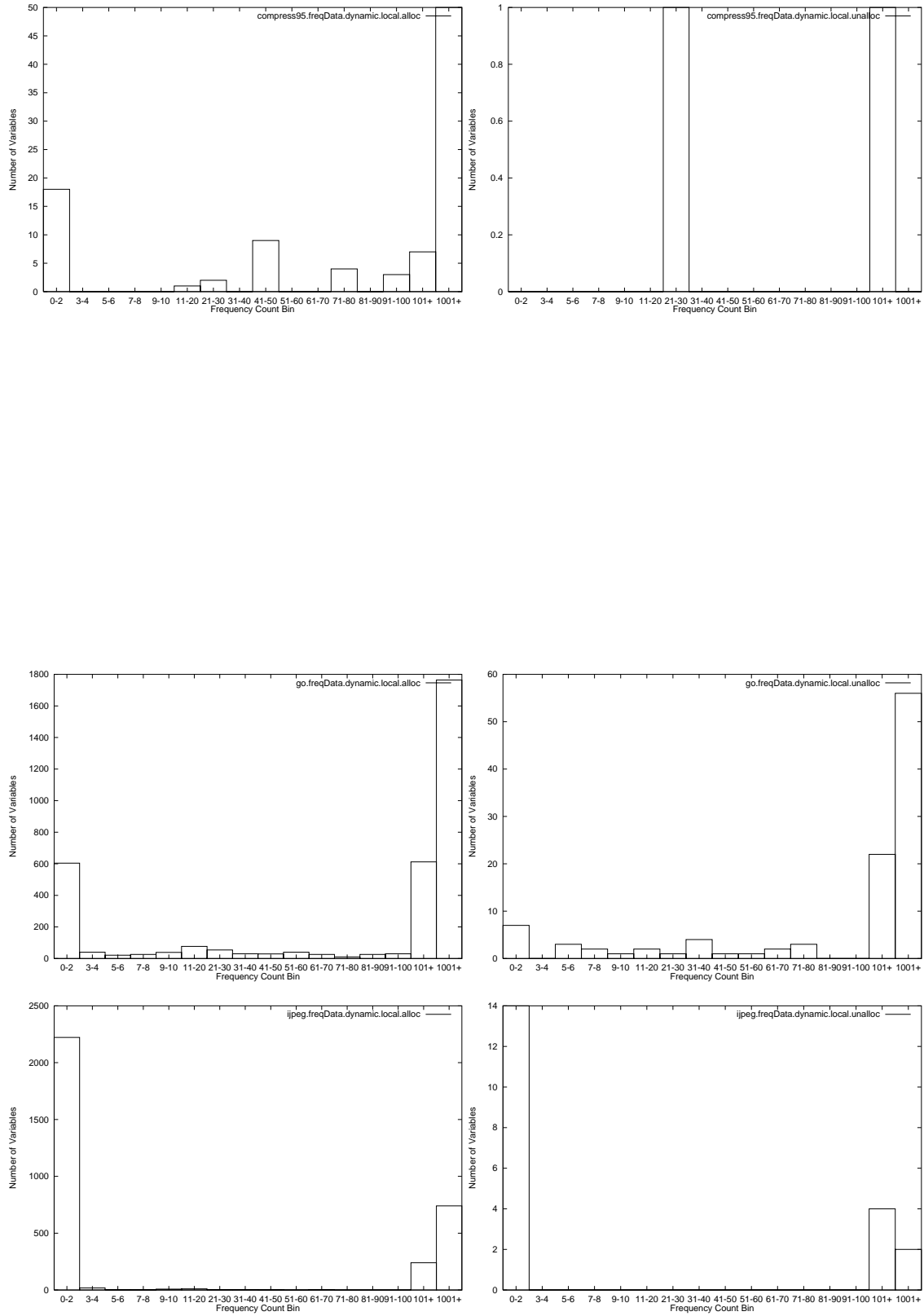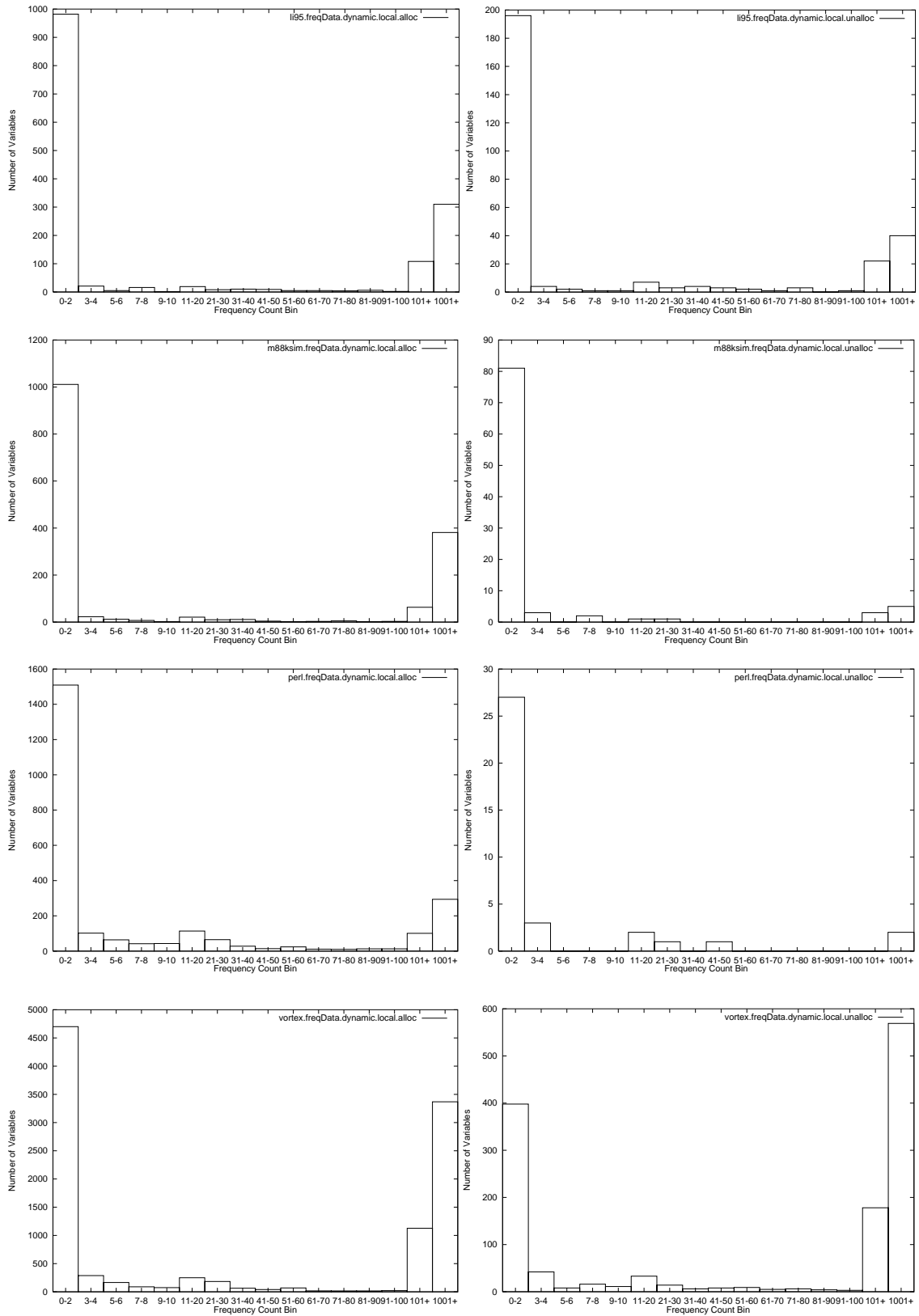**Figure 14 (cont). Static local variable frequency counts for SPECint95.**

Figure 15. Dynamic local variable frequency counts for SPECint95.

Figure 15 (cont). Dynamic local variable frequency counts for SPECint95.

Bibliography

[1]   David R. Ditzel and H. R. McLellan. Register Allocation for Free: The C Machine Stack Cache. Proc. Symp. Architectural Support for Programming Languages and Operating Systems, pp. 48-56, March, 1982.

[2]   D. A. Patterson and D. R. Ditzel. The Case for the Reduced Instruction Set Computer. can,, Vol. 8 No. 6, pp. 25-33. Oct, 1984.

[3]   D. W. Clark and W. D. Strecker. Comments on 'the Case for the Reduced Instruction Set Computer'. Computer Architecture News, Vol. 8 No. 6, pp. 34-38. Oct, 1980.

[4]   Scott A. Mahlke, William Y. Chen, Pohua P. Chang and Wen-mei W. Hwu. Scalar Program Performance on Muliple-Instruction-Issue Processors with a Limited Number of Registers. Proc. 25th Hawaii Intl. Conf. System Sciences, pp. ?-?, Jan, 1992.

[5]   H. Dietz and C.-H. Chi. CRegs: A New Kind of Memory for Referencing Arrays and Pointers. Proc., Supercomputing '88: November 14--18, 1988, Orlando, Florida, pp. 360-367, Jan, 1988.

[6]   S. Nowakowski and M. T. O'Keefe. A CRegs Implementation Study Based on the MIPS-X RISC Processor. Intl. Conf. Computer Design, VLSI in Computers and Processors, pp. 558-563, Oct, 1992.

[7]   Peter Dahl and Matthew O'Keefe. Reducing Memory Traffic with CRegs. Proc. 27th Intl. Symp. Microarchitecture, pp. 100-104, Nov, 1994.

[8]   Keith D. Cooper and Timothy J. Harvey. Compiler-Controlled Memory. Eighth Intl. Conf. Architectural Support for Programming Languages and Operating Systems, pp. 100-104, Oct, 1998.

[9]   Richard L. Sites. How to Use 1000 Registers. Proc. 1st Caltech Conf. VLSI, pp. 527-532, Jan, 1979.

[10]   Gary S. Tyson and Todd M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. Proc. 30th Intl. Symp. Microarchitecture, pp. 218-227, Dec, 1997.

[11]   Douglas C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin, Madison Tech. Report. June, 1997.

[12]   Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins and Peter W. Markstein. Register Allocation Via Coloring. Computer Languages, Vol. 6 No. 1, pp. 47-57. unknown month, 1981.

[13]   G. J. Chaitin. Register Allocation and Spilling via Graph Coloring. Proc. SIGPLAN '82 Symp. Compiler Construction, pp. 98-105, unknown month, 1982.

[14]   Preston Briggs. Register Allocation via Graph Coloring.. Rice University, Houston,

Texas, USA Tech. Report. unknown month, 1992.

[15] Manoj Franklin. The Multiscalar Architecture. University of Wisconsin, Madison Tech. Report. Nov, 1993.

[16] Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[17] Samuel P. Harbison. An Architectural Alternative to Optimizing Compilers. Proc. Symp. Architectural Support for Programming Languages and Operating Systems, pp. 57-65, March, 1982.

[18] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. Proc. ACM SIGPLAN'95 Conf. Programming Language Design and Implementation (PLDI), pp. 1-, , 1995.

[19] Rakesh Ghiya and Laurie J. Hendren. Putting Pointer Analysis to Work. Conf. Record POPL~'98: The 25th ACM SIGPLAN-SIGACT Symp. Principles Programming Languages, pp. 121-133, , 1998.

[20] Richard Hank. Machine Independent Register Allocation for the IMPACT-I C Compiler. University of Illinois at Urbana-Champaign Tech. Report. Jan, 1993.

[21] Vineeth Kumar Paleri, Y. N. Srikant and Priti Shankar. A Simple Algorithm for Partial Redundancy Elimination. ACM SIGPLAN Notices, Vol. 33 No. 12, pp. 35-43. Dec, 1998.

[22] D. W. Goodwin and K. D. Wilken. Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming. Soft\-ware\emdash Prac\-tice and Experience, Vol. 26 No. 8, pp. 929-??. Aug, 1996.

[23] Timothy Kong and Kent. D. Wilken. Precise Register Allocation for Irregular Architectures. Proc. ACM SIGPLAN'98 Conf. Programming Language Design and Implementation (PLDI), pp. 297-307, , 1998.

[24] Keith Cooper and John Lu. Register Promotion in C Programs. Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI-97), pp. 308-319, June, 1997.

[25] A. V. S. Sastry and Roy D. C. Ju. A New Algorithm for Scalar Register Promotion Based on SSA Form. Proc. ACM SIGPLAN'98 Conf. Programming Language Design and Implementation (PLDI), pp. 15-25, , 1998.

[26] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu and Peng Tu. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores. Proc. ACM SIGPLAN'98 Conf. Programming Language Design and Implementation (PLDI), pp. 26-37, , 1998.

[27] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. ACM Transactions Programming Languages and Systems, Vol. 12 No. 4, pp. 501-536. unknown month, 1990.

[28] August G. Reinig. Alias Analysis in the DEC C and DIGITAL C++ Compilers.

Digital Technical Journal, Vol. 10 No. 1, pp. 48-57. Dec, 1998.

[29] John Lu. Interprocedural Pointer Analysis for C. Rice University, Houston, Texas, USA Tech. Report. April, 1998.

[30] M. J. Charney and T. R. Puzak. Prefetching and memory system behavior in the SPEC95 benchmark suite. IBM Journal Research and Development, Vol. 41 No. 3, pp. 265-286. May, 1997.

[31] S. Bandyopadhyay, V. S. Begwani and R. B. Murray. Compiling for the CRISP Microprocessor. Proc. 1987 Spring COMPCON, pp. 265-286, Feb, 1987.

[32] D. R. Ditzel, H. R. McLellan and A. D. Berenbaum. Design Tradeoffs to Support the C Programming Language in the CRISP Microprocessor. Proc. Second Intl. Conf. Architectural Support for Programming Languages and Operating Systems-ASPLOSII, pp. 158-163, Oct, 1987.

[33] David R. Ditzel, Hubert R. McLellan and Alan D. Berenbaum. The Hardware Architecture of the CRISP Microprocessor. Proc. 14th Intl. Symp. Computer Architecture, pp. 309-319, June, 1987.

[34] A. D. Berenbaum, D. R. Ditzel and H. R. McLellan. Architectural Innovations in the CRISP Microprocessor. Proc. SPRING COMPCON87, pp. 91-95, Feb, 1987.

[35] A. D. Berenbaum, D. R. Ditzel and H. R. McLellan. Introduction to the CRISP Instruction Set Architecture. Proc. SPRING COMPCON87, pp. 86-90, Feb, 1987.

[36] Chen Ding. Improving Software Pipelining with Unroll-and-Jam and Memory Reuse Analysis. Master's Thesis, Michigan Technological University, Department of Computer Science, 1996.

[37] T. Suchyta. Global Reduction of Spill Code by Live-Range Splitting. Master's Thesis, Michigan Technological University, Department of Computer Science, 1998.

[38] David J. Kuck. The Structure of Computers and Computations. John Wiley & Sons, Pittsburgh, Pennsylvania, 1978.

[39] J. P. Anderson, S. A. Hoffman, J. Shifman and R. J. Williams. D825-A Multiple Computer System for Command and Control. Proc. AFIPS Fall Joint Computer Conf., pp. 86-96, unknown month, 1962.

[40] S. E. Gluck. Impact of Scratchpads in Design: Multifunctional Scratchpad Memories in the Burroughs B8500. Proc. AFIPS Fall Joint Computer Conf., pp. 661-666, unknown month, 1965.

[41] T. Kilburn, D. B. G. Edwards, M. J. Lanigan and F. H. Sumner. One-level Storage System. IRE Transactions Electronic Computers, Vol. EC-11 No. 2, pp. 223-235. April, 1962.

[42] Daniel P. Siewiorek, C. G. Bell and A. Newell. Computer Structures: Principles and Examples. McGraw-Hill, Pittsburgh, Pennsylvania, 1982.

[43] S. Takahashi, H. Nishino, K. Yoshihiro and K. Fuchi. System Design of the ETL Mk-6 Computers. Information Processing 1962, Proc. IFIP Congress, pp. 690-, unknown month, 1963.

[44]   P. G. Emma. Understanding some simple processor-performance limits. IBM Journal Research and Development, Vol. 41 No. 3, pp. 215-231. May, 1997.

[45]   Antonio Gonzalez, Jose Gonzalez and Mateo Valero. Virtual-Physical Registers. Proc. 4th Intl. Symp. High-Performance Computer Architecture (HPCA-4), pp. ??-??, Feb, 1998.

[46]   D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran and W. W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. Proc. 25th Intl. Symp. Computer Architecture, pp. 227-237, June, 1998.

[47]   Jim Pierce. IA-64 Architecture Innovations (Intel Corporation). Slides from a presentation at the University of Michigan, Feb, 1999.

[48]   Robert Yung and Neil C. Wilhelm. Caching Processor General Registers. Intl. Conf. Computer Design, pp. 307-312, Oct, 1995.

[49]   Robert Yung and Neil C. Wilhelm. Caching Processor General Registers. Sun Microsystems Laboratories Tech. Report. June, 1995.

[50]   Pritpal S. Ahuja, Douglas W. Clark and Anne Rogers. The Performance Impact of Incomplete Bypassing in Processor Pipelines. Proc. 28th Intl. Symp. Microarchitecture, pp. 36-45, Nov, 1995.

[51]   William Lonergan and Paul King. Design of the B5000 system. Datamation, Vol. 7 No. 5, pp. 28-32. May, 1961.

[52]   http://www.eecs.umich.edu/mirv.

[53]   Krisztian Flautner, Gary S. Tyson and Trevor N. Mudge. MirvSim: A high level simulator integrated with the Mirv compiler. Proc. 3rd Workshop Interaction between Compilers and Computer Architectures (INTERACT-3), pp. ?-?, Oct, 1998.

[54]   http://www.edg.com.

[55]   R. J. Hookway and M. A. Herdeg. DIGITAL FX!32: Combining Emulation and Binary Translation. Digital Technical Journal Digital Equipment Corporation, Vol. 9 No. 1, pp. 3-??. , 1997.

[56]   J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Mateo, CA, 1996.

[57]   Luiz Andre Barroso, Kourosh Gharachorloo and Edouard Bugnion. Memory System Characterization of Commercial Workloads. Proc. 25th Intl. Symp. Computer Architecture, pp. 3-14, June, 1998.

[58]   Gordon E. Moore. Cramming more components onto integrated circuits. Electronics, Vol. 38 No. 8, pp. 114-117. April, 1965.

[59]   Vinodh Cuppu, Bruce Jacob, Brian Davis and Trevor Mudge. A Performance Comparison of Contemporary DRAM Architectures. Proc. 26th Intl. Symp. Computer Architecture, pp. ??-??, June, 1999.

[60]   Motorola. M68000 Family Programmer's Reference Manual. Motorola, Phoenix,

AZ, 1992.

[61]   Intel. Intel Architecture Software Developer's Manual: Instruction Set Reference. Intel, Santa Clara, CA, 1997.

[62]   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. Proc. 20th Intl. Symp. Computer Architecture, pp. 247-256, May, 1993.

[63]   D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal and W.-M. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. ACM SIG-PLAN Notices, Vol. 29 No. 11, pp. 183-183. Nov, 1994.

[64]   Great Microprocessors of the Past and Present. http://www.microprocessor.sscc.ru/great.

[65]   Alexandru Nicolau. Run-Time Disambiguation: Comping with Statically Unpredictable Dependencies. IEEE Transactions Computers, Vol. 38 No. 5, pp. 663-678. May, 1989.

[66]   MicroDesign Resources. Chart Watch: Workstation Processors. Microprocessor Report, Vol. 13 No. 1, pp. 31. Jan, 1999.

[67]   Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. IEEE Micro, Vol. 16 No. 2, pp. 28-40. April, 1996.

[68]   Linley Gwennap. PA-8000 Combines Complexity and Speed. Microprocessor Report, Vol. 8 No. 1, pp. 1-5. Nov, 1994.

[69]   Standard Performance Evaluation Corporation. SPEC CPU95 Technical Manual. http://www.spec.org, Manassas, Virginia, 1995.

[70]   David W. Wall. Global Register Allocation at Link Time. Proc. SIGPLAN'86 Symp. Compiler Construction, pp. 264-275, July, 1986.

[71]   M. V. Wilkes. Slave Memories and Dynamic Storage Allocation. IEEE Transactions Electronic Computers, Vol. EC-14 No. 2, pp. 270-271. April, 1965.

[72]   Alan Jay Smith. Cache Memories. ACM Computing Surveys, Vol. 14 No. 3, pp. 473-530. Sep, 1982.

[73]   John A. Swenson and Yale N. Patt. Hierarchical Registers for Scientfic Computers. Proc. Intl. Conf. Supercomputing, pp. 346-353, July, 1988.

[74]   Andrew Ayers, Richard Schooler and Robert Gottlieb. Aggressive Inlining. Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PL-DI-97), pp. 134-145, June, 1997.

[75]   J. S. Emer and D. W. Clark. A Characterization of Processor Performance in the VAX- 11/780. Proc. 11th Symp. Computer Architecture, pp. 301-310, June, June 1984.

[76]   David L. Weaver and Tom Germond. The SPARC Architecture Manual, Version 9. Sparc International and PTR Prentice Hall, Englewood Cliffs, NJ, 1994.

[77]   IBM. Enterprise Systems Architecture/390 Principles of Operation. IBM, Pough-keepsie, New York, 1998.

[78]   Mary W. Hall, Jennifer M. Anderson, Saman p. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. Computer, Vol. 29 No. 12, pp. 84-??. Dec, 1996.

[79]   Digital Equipment Corporation. The PDP-11 Architecture Handbook. Digital Press, Burlington, MA, 1983.

[80]   Richard A. Brunner. The VAX Architecture Reference Manual. Digital Press, Burlington, MA, 1991.

[81]   Richard L. Sites. The Alpha Architecture Reference Manual. Digital Press, Burlington, MA, 1992.

[82]   Brian Case and Michael Slater. DEC Enters Microprocessor Business with Alpha. Microprocessor Report, Vol. 6 No. 3, pp. 24. March, 1992.

[83]   R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM Journal Research and Development, Vol. 11 No. 1, pp. 25-33. Jan, 1967.

[84]   Robert M. Keller. Look-Ahead Processors. ACM Computing Surveys, Vol. 7 No. 4, pp. 177-195. Dec, 1975.

[85]   Frederick Chow and John Hennessy. Register Allocation by Priority-based Coloring. ACM SIGPLAN Notices, pp. 222-232, June, 1984.

[86]   David S. Blickstein, Peter W. Craig, Caroline S. Davidson, R. Neil Faiman, Jr., Kent D. Glossop, Richard B. Grove, Steven O. Hobbs and William B. Noyce. The GEM Optimizing Compiler System. Digital Technical Journal Digital Equipment Corporation, Vol. 4 No. 4, pp. 121-136. , 1992.

[87]   Maryam Emami, Rakesh Ghiya and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. Proc. SIGPLAN '94 Conf. Programming Lanuage Design and Implementation, pp. 242-256, , 1994.

[88]   UNIX System Laboratories Inc.. System V Application Binary Interface: MIPS Processor Supplement. Unix Press/Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[89]   I-Cheng K. Chen, Peter L. Bird and Trevor Mudge. The Impact of Instruction Compression on I-cache Performance. Computer Science and Engineering, University of Michigan Tech. Report. Feb, 1997.

[90]   B. Heggy and M. L. Soffa. Architectural Support for Register Allocation in the Presence of Aliasing. Proc., Supercomputing '90: November 12--16, 1990, New York Hilton at Rockefeller Center, New York, New York, pp. 730-739, Feb, 1990.