Register Allocation for VLSI Processors

by

Wei-Chung Hsu

Computer Sciences Technical Report #619

November 1985

Register Allocation for VLSI Processors

Wei-Chung Hsu

Computer Sciences Department
University of Wisconsin — Madison
Madison, WI 53706

Abstract

The performance of a single-chip CPU is often restricted by limited off-chip memory bandwidth. In this report, we study how to effectively use registers – one kind of on-chip memory – to reduce off-chip memory accesses. To use a limited number of registers efficiently, a good register allocation should handle spilling effectively. We study the minimization of Load/Store instructions in local register allocation. Both an optimal and an efficient heuristic algorithm are proposed for local register allocation. We also suggest the use of a trace optimization technique for global register allocation.

This research was supported by the National Science Foundation under grants MCS82-02952 and DCR-8105904

1. Introduction

The performance of a single-chip CPU is often restricted by limited off-chip memory bandwidth [Hill85]. In this report, we study how to use registers effectively, which is one kind of on-chip memory, to reduce off-chip memory accesses. To use a limited number of registers efficiently, a good register allocation should handle spilling effectively. In this work, we study the minimization of Load/Store instructions in local register allocation. We also suggest using a trace optimization technique for global register allocation.

In section 2, we compare registers with data caches. Allocation of variables in registers often increases the overhead of procedure calls. Although in-line expansion techniques can reduce procedure call overhead, they shift register save/restore overhead to register allocation. In section 3, we discuss the issues of in-line expansion and its relationship to register allocation. We also measure the possible performance of a perfect register allocation. Section 4 describes the minimization of number of Load/Stores in local register allocation. Section 5 extends the allocation algorithm in section 4 to global allocation. The last section provides summary comments.

2. On-chip Memory

Advances in semiconductor technology have made it possible to design and fabricate an extremely high-performance CPU on a single chip. Two problems may prevent such a CPU from being effectively used: (1) The CPU may spend a large amount of time waiting for slow storage; (2) The CPU may require greater (off-chip) communication bandwidth than is available with current packaging technology. (Though new packaging techniques expand the number of pins, these may decrease reliability and increase power consumption [Goodman85b].)

In VLSI, the off-chip communications are both expensive and slow, while on-chip communications are inexpensive and fast [Patterson80, Hill84]. A successful VLSI architecture will probably include features that make efficient use of on-chip memory. In other words, effective use of local memory is a critical aspect of the design of a VLSI processor architecture.

Patterson has suggested a local memory hierarchy in a single-chip processor [Patterson80]. A memory hierarchy is effective because it has two kinds of locality of memory references: (1) spatial locality and (2) temporal locality. Spatial locality refers to the observation that memory locations near a location just accessed are likely to be accessed again. Temporal locality refers to the observation that references to a given location tend to be clustered in time. Spatial locality is particularly important when a new process is initiated because it can be exploited to fetch the working set quickly. However, fetching memory before it is requested increases the bandwidth requirement of the memory system. Since a major performance limitation of VLSI processors is the off-chip bandwidth, temporal rather than spatial is the locality of choice for examination.

A set of registers and a cache are two kinds of local memory. We distinguish between cache memory, which is a redundant subset of the memory system, and registers, which must be explicitly loaded by the processor. Since instructions exhibit temporal and spatial locality in a far more consistent manner than data, since an instruction cache is easier to implement than a data cache, and since an instruction cache does not have the coherence problem that a data cache does (in a multiprocessor system), there is a consensus that an on-chip instruction cache is a cost effective way to increase the performance of the CPU [SmithJ83, Moto82, Patterson83,Alpert83, Henessy84]. Given this consensus, to compare registers with a data cache is reasonable. A data cache is good because (1) it (nearly) always works reasonably well [SmithA85]; (2) it takes dynamic program behavior into account; and (3) it is invisible to programmers. A set of registers has different advantages: (1) it outperforms a data cache by a factor of two in both speed and cost [Ditzel82]; (2) it can exploit temporal locality by careful scheduling at compile time (in fact it does better than temporal locality: a compiler can look ahead and fetch data items into register before they are needed); (3) it typically requires fewer bits to address a register than a memory location, so instructions can be compact. Which one, cache or registers, is more suitable for VLSI processors? If the dynamic behavior of a program can be taken into account at compile time, then with careful scheduling, registers are more suitable than cache memories for VLSI processors.

One strong argument for cache memories is that they are architecture-independent. Therefore, they can be used to increase system performance without affecting programs. However, in order to increase performance, some computers make the cache visible. For example, the IBM 801 has instructions to allocate and free cache lines. Also, system programmers make the control program share the same stack with user programs to increase the cache hit ratio [Radin82]. For high-performance computing, it may be advantageous to make caches architectural.

One criticism of using many registers is that they are not easy to use effectively [Myers81]. However, an optimizing compiler can use them effectively [Radin82, Auslander82]. In addition, current high-performance architectures are designed with optimizing compiler techniques in mind [Hennessy81, Patterson82, Radin82, SmithJ83]. If the optimizing compiler can take full advantage of a large number of registers, they can be extremely useful. For example, the CRAY-1 computer has no data cache; instead, it uses numerous registers. The CFT (CRAY-1's FORTRAN compiler) can use those registers to effectively decrease the memory bandwidth requirement (programmers can do even better).

One major disadvantage of registers is the so called semantic gap [Myers81]. Registers are not easy to allocate for objects requiring multiple storage units (e.g. arrays, strings, records), while cache memories need no special treatment. Also, the aliasing problem keeps many data items from being allocated in registers. However, improving compiler techniques [Fisher84, Cooper84, Aho77] can reduce the frequency of unresolved aliases. Frances E. Allen, a well-known researcher in optimizing compilers has said, "It seems unlikely that new languages will continue to contain constructs such as unconstrained aliasing which confound analyses" [Allen81]. We believe that in the future, more and more data items will be allocatable in registers, so that registers may be effectively used to reduce off-chip bandwidth requirements.

3. Using Registers to Reduce LOAD/STORE Instructions

3.1. Register Allocation And Register Assignment

The phrase "register allocation" is ambiguous. It has been used to describe a number of different problems and activities such as code generation techniques [Aho72, Sethi70, Sethi75] and optimizations [Chaitin81, Chaitin82, Chow84, Harrison75, Sites79]. People usually use "register allocation" and "register assignment" to distinguish between two phases of compilation as follows :

Allocation is the decision of what names in a program should reside in registers. For a register-oriented architecture, this includes the following:

(1) Finding registers in which to do arithmetic and logic operations. In an architecture that accesses memory only with LOAD/STORE instructions, all sources and destinations must be registers.

(2) Finding locations in which to save the results of operations for later use. The value of a subexpression, for example, is usually saved in a register.

(3) Passing of parameters to and returning results from subroutines.

(4) Using registers as a fast buffer, so that frequently-used variables can be accessed quickly. This reduces the number of memory accesses.

We will focus primarily on the use of registers as fast buffers in this proposal.

Assignment is the association of preallocated symbolic registers with real registers. To simplify the IL (Intermediate Language or Internal Language) code generation, many compilers [Auslander82, Powell84, Kim78, Leverett81] initially assume a hypothetical target machine having an unlimited number of general purpose high speed registers. The optimizer takes advantage of this assumption to eliminate references to memory by keeping data in registers as much as possible. The assignment phase then maps the unlimited number of virtual registers to the limited number of real registers, spilling where necessary.

The responsibility of the allocation phase is to allocate as many data items as possible to symbolic registers; the responsibility of the assignment phase is to assure that the number of additional LOAD/STOREs generated for spilling is as small as possible.

My work is primarily on developing a good assignment algorithm. Much previous work [Chaitin81, Chaitin82, Chow84, Kim78, Sites79] used the term "allocation" for register assignment; I will continue to use "register allocation" for "register assignment" in order to maintain consistency with the literature.

3.2. Register Allocation and Procedure Calls

Register allocation has long been considered a difficult problem [Allen81]. Existing compilers may use simple algorithms, but experimental compilers often test more complicated ones [Auslander82, Allen80, Chow83, Levertee81]. In order to take full advantage of fast registers and effectively use a large number of registers, ambitious optimizing compilers keep looking for a better allocation algorithm.

Recently, an elegant approach has been developed and implemented by G. Chaitin, primarily for the PL.8 compiler in the IBM 801 project [Chaitin81, Auslander82, Radin82]. In this approach, the problem is formulated as a graph coloring problem: Each node in the graph stands for a computed quantity that resides in a machine register, and two nodes are connected by an edge if the quantities interfere with each other (*i.e.*, if they are simultaneously live). The goal is to assign different colors (registers) to connected nodes. When the compiler cannot color the graph with a number of colors equal to the number of available registers, it must add code to store and reload register contents to and from storage. The implementation showed that a fast heuristic method for assigning colors to these particular graphs generally resulted in a very good assignment. A later algorithm (developed at Stanford and used in the MIPS project) is based on the same model but has an enhanced spilling algorithm as a supplement [Chow84].

Both graph coloring algorithms claim to work very well: rarely is there need for code spilling [Radin82, Chow84]. However, Patterson has claimed

> About 30 percent of the 801 instructions are LOAD or STORE when large programs are run; the MIPS has 16 registers compared to 32 for the 801, about 35 percent of them being LOAD or STORE instructions. For the Berkeley RISC machines, this percentage drops to about 15 percent, including the LOADs and STOREs used to save and restore registers when the register-window buffer overflows. [Patterson85]

If the graph coloring algorithm requires little spilling code, why is the number of LOADs and STOREs so high? Apparently, because of procedure calls [Hennessy84].

Normally, register allocation is done only within a procedure. If more registers are used in a procedure, then the opening cost of the procedure is higher: More STOREs/LOADs are required when this procedure is being called or returned from. Register allocation increases the cost of procedure calls. This important observation has been reported many times [Ditzel82, Hennessy84, DEC82]. Some architectures have special instructions to support procedure calls. The VAX, for example, has the CALLS instruction, which performs all actions needed by a procedure call (including register save/restore). However, because of the slowness of the CALLS instruction, some compilers (*e.g.*, the MODULA-2 compiler written by M. Powell) choose to use more primitive instructions (such as JMP/RBS and MOV) instead [Patterson85]. There are almost no LOADs and STOREs associated with procedure calls in RISC I, but many LOADs and STOREs are needed to save and restore registers in the 801 and MIPS. This is probably a major reason that there are so many LOADs and STOREs in the dynamic trace of the 801 and MIPS.

A simple procedure call in compiled code without register allocation is not very expensive: the PC must be saved, the old activation record pointer must be saved, a new activation record must be created, and in block-structured language, the display must be updated. When register-allocated variables need to be saved and restored, the cost of procedure call rises rapidly. Because of the register saving/restoring overhead, procedure calls become the most costly source-language statement [Patterson82]. In order to speed up procedure calls, some architectures have included a register stack [Ditzel82], or overlapped register windows [Patterson82, Ragan83]. Though multiple register sets

are effective for eliminating the LOAD/STORE overhead on procedure calls, this technique introduces new problems: The large set of registers may slow down basic cycle time, it consumes a large area of silicon, and it increases process-switching time.

Patterson has claimed that if compiler technology can reduce the number of LOADs and STOREs to the extent that register windows can, an optimizing compiler will be clearly superior to a multiple register window scheme [Patterson85]. We are interested in knowing whether compiler technology for reducing LOADs/STOREs can be improved so that architectural support is not necessary.

Modern compilers use procedure integration, or *in-line expansion* to reduce procedure call overheads [Allen80, Auslander82, Madhavji82, Chow83, Maclaren84]. Procedure integration is a good program optimization technique though it may increase program size. It can reduce procedure call frequency, create larger basic blocks, remove parameter passing overhead, remove dead code and perform some computation at compile time through constant propagation [Allen72, Allen80, Scheifler77, Maclaren84, Ball79], and allow better global optimizations. The RISC (Reduced Instruction Set Computer) approach is basically a variation of *in-line expansion*, since compiling a program down to micro-instruction level is like expanding a complex instruction's microcode in-line. Although in-line expansion may increase the code space (if procedures are called more than once in the program text), it usually increases only the static code size, not the dynamic memory bandwidth. (The latter is what we are especially concerned with in designing VLSI processors.) In addition, modern programming practice encourages the use of many small procedures, many of which may be called exactly once. If most of the procedures in a program are just called once in the text, in-line expansion will decrease rather than increase the code space (because the instructions for prologue and epilogue can be removed) [Scheifler77]. When an on-chip instruction cache is used, in-line expansion must be done more carefully, since, if a procedure is called more than once in a loop, the loop, after expansion, may no longer fit in the cache. Cache misses increase the off-chip memory traffic. This is the reason that RISC people suggest implementing frequently used functions as pro-

cedures and executing them from the instruction cache.

Although procedure integration is a useful optimization technique, its implementation is not trivial. For instance, expansion order, maintenance of correct semantics, and determining when it is beneficial, must all be solved for procedure integration to be effective [Allen80]. In addtion, there are two other problems with in-line expansion : Recursive procedures and separate compilation. These two problems would not significantly restrict in-line expansion, because (1) there are techniques to transform a recursive procedure into a nonrecursive or iterative ones [Bird77]; (2) recursive procedures are not used frequently [Madhavji82]; (3) separate compilation is primary for program developing, and after a program is tested and debugged, its small components can be merged into a single program.

Procedure integration can reduce the number of procedure calls, but it cannot remove the register save/restore overhead. In-line expansion will create many more local variables in the calling procedure, and increase the degree of their interference. More registers are required to color the more complex interference graph. When compilers run out of registers, they must resort to register spilling. Therefore, rather than eliminating register save/restore overhead, it may merely shift the problem from procedure calls to spilling. Alternatively, it allows the compiler to make effective use of more registers, *if they are available* [Radin82].

One weak point of the graph coloring algorithm is that it can not handle spilling very well. The spilling process that it uses is slow and may generate many more LOADs/STOREs than necessary. For example, when the PL.8 compiler generated code for some complex programs for the MC68000, the excessive spill code made the code space much larger. When the coloring algorithm is blocked, it will spill the node for which the cost of spilling it is the smallest (where the cost is defined as estimated number of uses of that node). This process continues spilling node by node until the interference graph can be colored. This method has two drawbacks: (1) the estimated cost of each node may be misleading since, at run time, some branches are traversed more frequently than others; (2) spilling the node with least cost may not lead to the minimal number of

LOAD/STOREs. Sometimes, spilling a combination of two or more nodes may cause fewer LOAD/STOREs. This can be illustrated in the following example.

	A	B	C	D	E
1	def				def
2				def	
3	use			use	
4	.			.	
5	.			.	
6					use
7					def
8		def			
9		use			
10		.			
11					use
12					def
13			def		
14			use		
15			.		
16					use
17				use	

figure 3.1

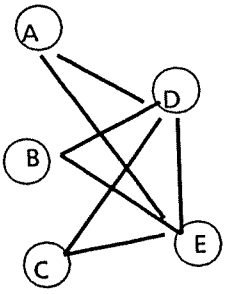

figure 3.2

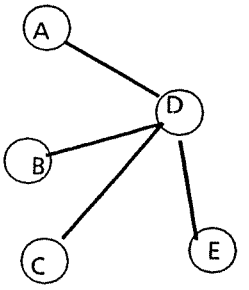

figure 3.3

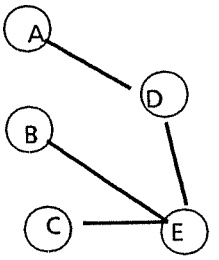

figure 3.4

In figure 3.1, the reference pattern is given for variables A, B, C, D and E; a "." means several clustered uses. The interference graph is given in figure 3.3. Assume that only two registers are available. Figure 3.2 can not be colored with two colors; thus spilling is necessary. Now suppose that E is the node with least cost to spill; spilling E may change the graph from figure 3.2 to figure 3.3, which *can* be colored with two colors. It takes three STOREs and three LOADs to spill E. If we spill E from location 1 to 6, and spill D from location 6 to 17 (the interference graph as in figure 3.4), the graph is again 2-colorable, and it only takes two STOREs and two LOADs.

If we want to minimize the number of LOAD/STOREs inserted, a large number of possible combinations must be tried. This requires an infeasible amount of computation for a slightly larger example. Spilling is a difficult problem for the graph coloring algorithm, and the performance of the heuristic algorithm for spilling is questionable.

Chaitin [Chaitin81] claims this is not a problem, because (1) spilling occurs rarely; and (2) spilling converges quite rapidly. However, if we want to reduce the overhead of procedure calls by procedure integration, then many many more variables will interfere with each other. The interference graph becomes more complex, which makes the spilling process slow and results in poor code.

Both the 801's PL.8 compiler and MIPS's UOPT optimizer used procedure integration to reduce procedure call frequency. But the number of LOAD/STORE instructions is still high compared to RISC I [Patterson85]. From the above discussion, we feel that allocation algorithms based on graph coloring will not work well with procedure integration techniques.

The graph coloring algorithm is elegant, since it handles machine idiosyncrasies (*e.g.*, register pairs, dedicated registers) uniformly and systematically. However, if the goal is to minimize the number of LOAD/STORE instructions, perhaps we should consider a different algorithm which can handle spilling effectively.

3.3. Register Allocation Using MIN

3.3.1. Introduction

Before we start to work on a different register allocation algorithm, it is instructive to examine the performance of an optimal register allocation and the degree of performance degradation resulting from the conventional allocation algorithm. If current algorithms have near-optimal performance, further research in this area may not be worthwhile. In this study, we use a LOAD/STORE architecture and define a "perfect" register allocation as follows:

(1) Registers can be dynamically allocated at run-time.

(2) Future memory reference information is known.

(3) There are no semantic restrictions; every memory word can be allocated in a register.

3.3.2. The performance of the register allocation using MIN

To measure the performance of a "perfect" register allocation, we use a simulator which is basically a cache simulator with the following characteristics:

(1) It is fully associative;

(2) It uses a Write Back strategy;

(3) The transfer unit is equal to the addressing unit, which is one word, *i.e.* one register; and

(4) The replacement algorithm is Belady's MIN [Belady66].

There are two important performance metrics: *miss ratio* and *bus traffic*. It is more precise to use the *bus traffic* as the performance measurement because we are trying to minimize the number of LOAD/STORE instructions by register allocation. However, measuring the optimal bus traffic is much harder than measuring the optimal miss ratio. (As will be shown in section 4, only non-polynomial time algorithms for minimizing bus traffic are known, and these are incapable of handling the millions of memory references in a normal trace) On the other hand, the Belady's MIN algorithm is simple, fast, and optimal on minimizing the miss ratio. Therefore, we choose to use the miss ratio as the performance indicator. This is reasonable, since a lower miss ratio often means less bus traffic.

We generated and used 5 different traces for a VAX-11 architecture running UNIX version 4.2bsd:

SORT The standard UNIX sorting program sorting the first 1000 entries in the dictionary.
GREP The UNIX string matching program, searching through the dictionary for a string match.
NROFF The UNIX text formatter interpreting the Berkeley macro package -me.
CACHE A trace-driven cache simulator program simulating a fully associative, write-back cache.
AS The standard UNIX (VAX-11) assembler translating an assembly program jmalloc.s.

The input traces include only data fetches and stores. Some local variables may be allocated in registers by the compiler (because of register use "hints" in C); we removed all register hint declarations

from the source programs in order to obtain more realistic memory reference patterns. (The other approach is to treat registers as a memory extension, thus including all register accesses as memory references. This approach is less precise since one data item may have two addresses.)

Miss Ratio of "Perfect" Register Allocation (percent)				
Input	cache sizes (words)			
Trace	4	8	16	32
SORT	33.30	16.88	5.38	2.22
GREP	24.42	9.02	4.06	0.48
CACHE	34.13	19.95	9.50	7.95
AS	42.28	24.71	9.89	5.68
NROFF	44.32	26.58	10.36	4.56
Avg	35.69	19.43	7.83	4.18

Table 3.1

From table 3.1, we see that a small (fewer than 32) element register file may have a very high hit ratio if there is "perfect" register allocation.

For this experiment, "perfect" register allocation can achieve the minimal miss ratio, not the minimal bus traffic. We are interested in knowing how good the "perfect" register allocation is at decreasing bus traffic. From our experiment, on average, there are 190,000 memory references in a trace. The "perfect" register allocation results in about 35,000 memory accesses with 8 registers. This means it can decrease the memory accesses by about 80%. The percentage is 90% with 16 registers and 92% with 32 registers. If the register "hints" were used for the most frequently used variables (only 6 registers available for allocation in VAX-11, we modified the assembly programs to make use of 8 registers), the UNIX C compiler can decrease the number of memory accesses by 40% for the *sort* program, 53% for the *grep* and 32% for the *cache*. The MODULA-2 compiler [Powell84] has more optimization functions than that of the UNIX C compiler. It uses registers for subexpression temporaries, loop indices , loop limit values and scalar variables. It can decrease the number of memory accesses by 50% for the *sort* program (we rewrote the *sort* program in MODULA-2). Apparently, the "perfect" register allocation can decrease significantly more bus traffic than conventional compiler can. Further, as will see in the next chapter, "perfect" allocation

can perform even better by saving unnecessary stores. There is a substantial gap between the performance of the "perfect" allocation and current techniques.

3.4. Local access and Global access

Both the overlapped register window approach of RISC I and the register stack approach in the C stack machine [Ditzel82] eliminate LOAD/STOREs from local (stack) accesses. In order to compare our approach with those two, we split data fetches into two independent streams: stack accesses and global accesses. Simulations for "perfect" register allocation were done for each stream.

Miss Ratio of "Perfect" Register Allocation Stack Access only					
Input Traces	cache sizes (words)				
(% of all data accesses)	2	4	8	16	32
SORT(45%)	84.47	67.73	34.46	2.35	0.9
GREP(75%)	84.56	71.71	47.96	20.32	0.03
CACHE(41%)	34.26	16.15	10.53	4.93	1.44
AS(62%)	66.52	54.43	33.73	7.87	2.30
NROFF(78%)	79.01	63.94	34.29	10.43	1.65
Avg	69.76	56.79	32.19	9.18	1.26

Table 3.2

Miss Ratio of "Perfect" Register Allocation Global Access only					
Input Traces	cache sizes (words)				
(% of all data accesses)	2	4	8	16	32
SORT(55%)	47.65	25.72	10.64	8.89	8.41
GREP(25%)	33.16	18.41	8.39	3.43	3.25
CACHE(59%)	69.48	53.70	33.73	32.63	31.39
AS(38%)	51.43	37.35	30.43	22.00	18.00
NROFF(23%)	68.79	59.87	47.45	27.73	14.6
Avg	54.10	39.01	26.13	18.93	15.13

Table 3.3

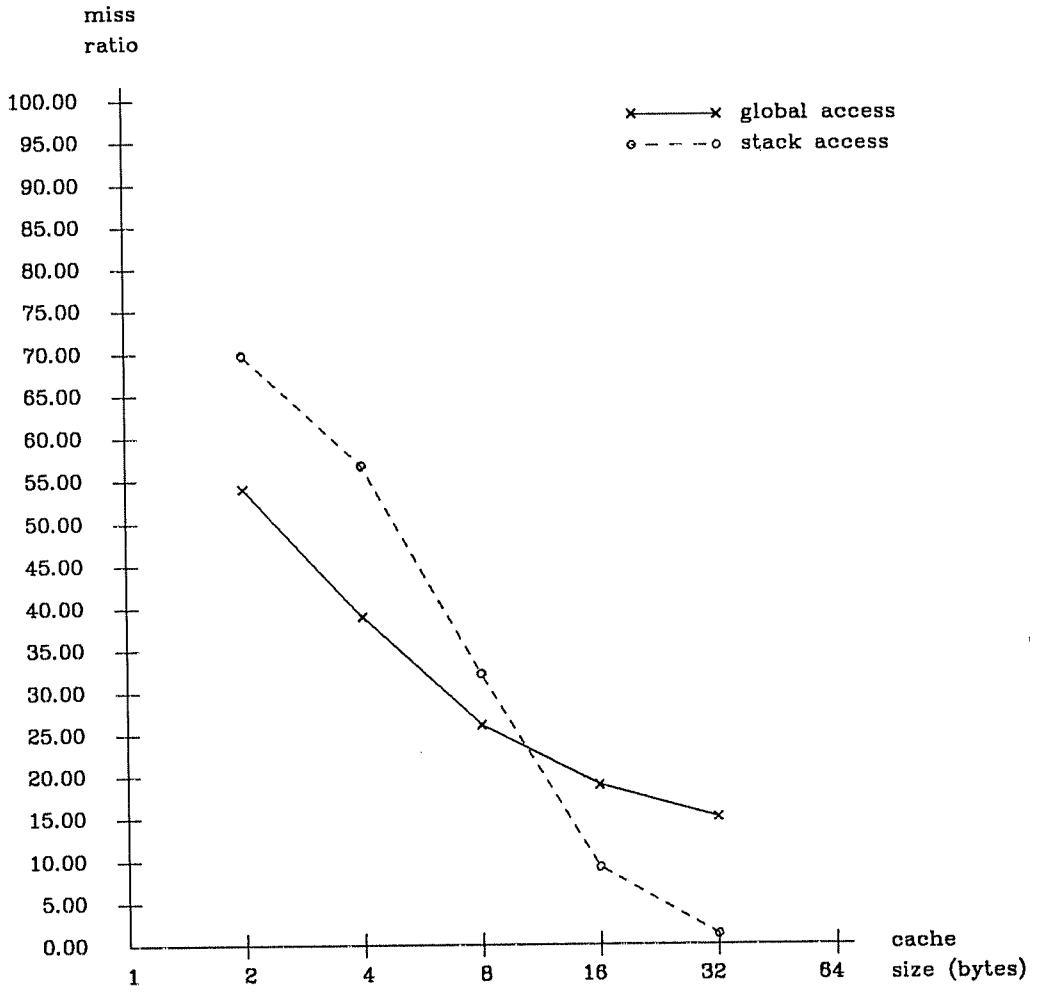

Figure 3.5

Table 3.2 shows that, with 32 registers, most local data items can be accessed from registers, using "perfect" register allocation. It is thus possible to eliminate most memory accesses by allocating local data items in registers. The register window approach may achieve similar results, but it is inefficient because it consumes a large chip area, only a small portion of which is in use at any time [Hennessy84]. As for global accesses, Ditzel [Ditzel82] suggests using a small number of registers allocated for the most frequently used scalar variables. This is because a few global scalar variables account for 90% of the dynamic references [Ditzel82]. Our results are compatible with his suggestion. In our experiments, the programs *compact*, *cache* and *sort* have 1198, 7658 and 10218 different global locations referenced, respectively, of which 60% of the dynamic references occur in 12, 7 and 10 locations. This "90-10" property (programs spend 90% of the time in 10% of the code) shows Ditzel's suggestion is a cost-effective way to use registers.

3.5. Discussion

In the previous section, we learned that, given an optimal register allocation , a register file with 32 registers can be extremely effective in reducing memory accesses from the *stack*. Nevertheless, there are several restrictions that normally prevent the realization of perfect performance:

(1) The semantic gap. Non-scalar variables cannot, typically, be allocated in registers. For example, arrays and strings must be assigned to primary memory. Because almost all of the arrays or structures are global variables (as shown in [Patterson82]), the semantic restrictions are not severe for local variables.

(2) Aliasing problems. If a data item can be reached through different names (or pointers) it is restricted to memory.

(3) Dynamic allocation. Once the instruction has been generated, the register designation cannot be changed at run time. Variables are bound to registers statically at compile time.

(4) Future reference information. Due to conditional branches and loop structures, only a limited amount of information concerning future reference can be obtained at compile time. Branch prediction may increase the available information concerning future reference: If the branch prediction is correct, then we can construct a compile time trace comparable to the run time trace.

Although it is difficult to attain performance that approaches that of the "perfect" allocation; compiler techniques are improving, anti-alias analysis has been developed [Fisher84], new programming languages such as Euclid [Popek77], Ada [Ledgard81] are designed to restrict uncontrolled aliases, and research on branch predictions is being conducted [Fisher83, Lee84, SmithJ81]. We therefore believe that the gap between optimal performance and the state of the art can be substantially reduced in the near future.

Recent developments in VLSI and high speed memories have encouraged computer architects to use large numbers of registers [Sites79a, Dannen79]. While architectures with large numbers of

registers can simplify register allocation (because a less complex allocation algorithm can be used, e.g. reference count), they have other deficiencies. A machine with a large number of registers has a slower basic clock cycle due to the capacitive loading of the longer bus and requires a longer instruction format. Also, increasing the number of registers slows down process switching. When the same allocation algorithm is used, increasing the number of registers beyond 32 results in only slightly improved performance (see figure 3.5). Although some compilers may use a large number of registers to reduce memory latency (using code scheduling techniques [Hennessy83]), this can also be achieved by using architectural queues [Young85].

4. Local Allocation

4.1. Introduction

Until now, little consideration has been given to minimizing the number of LOADs and STOREs by register allocation in straight-line programs (or basic blocks). This may be due to the fact that basic blocks are usually small. Thus, most of the well-known heuristic algorithms [Backus57, Freib74, Aho77, Kim78 Fischer82] have near-optimal results. However, it is important to reexamine this problem since many modern compiler techniques can generate large basic blocks [*e.g.* Fisher81].

In 1966, Horwitz *et al* [Horwitz66] published a definitive paper on index register allocation in straight-line programs, suggesting an *optimal*, non-polynomial-time algorithm. Their algorithm minimizes the number of loads and stores. Later algorithms [Luccio67, Kennedy72] are mainly variants of Horwitz's. However, those algorithms are concerned with index registers rather than general purpose registers. In Horwitz's model, two basic operations are defined for index registers: Read and Modification. A modification is, basically, a read followed by a write. Since the basic operations for general purpose registers are read and write (For example, the instruction "MOV r_1, r_2" reads register r_1 and writes register r_2; the instruction "ADD r_1, r_2, r_3" reads registers r_1 and r_2 and writes register r_3), Horwitz's algorithm is inappropriate for general purpose register alloca-

tion. We will discuss this problem further in section 4.2 where we define the cost function for general purpose register allocation.

We have constructed a model for optimal general purpose register allocation. We provide rules to speed up the computation of an optimal algorithm. We have devised a linear time heuristic algorithm which has performance approaching the optimal one. Finally, we compare this algorithm's performance with that of other algorithms.

4.2. The Model

Let us begin with some definitions.

$X = \{r_1, r_2, \ldots, r_M\}$ is a finite set, the set of variables.

Here, "variables" means "symbolic registers." We assume that, at the allocation phase, symbolic registers are assigned to local variables, temporaries, frequently used constants, etc. In the assignment phase, those symbolic registers are mapped to real registers. In this model, there are M variables and N real registers. A symbolic register (or variable) has a permanently associated memory location to store its content when it is spilled.

$S = \{s_1, s_2\}$ is a set of two states of variables; s_1:clean state, s_2:dirty state.

$Q_i = (q_1, q_2 \ldots, q_N)$ is a register configuration, which is an unordered set of N registers, where q_i belongs to $X \times S$. (r, s_1) means the value of variable r in a register is consistent with the value of r in memory -- it is clean. (r, s_2) means the value of variable r in a register is not consistent with its value in memory -- it is dirty.

The difference between a dirty variable and a clean variable is that if we wish to remove a dirty variable from the register configuration, we must update its memory location. This requires a "store". No store is required for removing a clean variable. However, if a dirty variable is dead, it need not be written to memory.

P = $\{I_1, I_2, \ldots, I_n\}$ is a program, a finite sequence of elements of X × S. A pair (r, s_1) in P represents a read of variable r while (r, s_2) represents a write of variable r. Therefore, a program here means a sequence of variable reads and writes. The *i*th symbol of the program is called the *i*th step. There are n steps in a program. For convenience, we use r to denote the pair (r,s_1), and r* to denote the pair (r,s_2).

We can specify an example program as follows :

$r_1^*\ r_0\ r_2^*\ r_1\ r_2\ r_3^*\ r_0\ r_4^*\ r_3\ r_4\ r_5^*\ r_0\ r_5$

The above program is a register access trace from the following sample program segment:

```
L  r1,0
L  r2,X(r0)
A  r1,r2,r3
L  r4,Y(r0)
A  r3,r4,r5
ST r5, Z(r0)
```

The model we have defined is not a realistic one in that it does not deal with multiple register reads and writes in a single step. However, this simplified model can help study how to make good replacement decisions, and can be adjusted when necessary.

$A = (Q_1, .., Q_n)$ is an allocation for a program P, which is a sequence of register configurations. To be a legal allocation for P, the following condition must hold:

$$I_i \in Q_i \text{ , for every i, } 1 \leq i \leq n$$

If the variable accessed at *i*th step is in the configuration, it is said to be a hit, if not, it is said to be a miss. We will call it a write miss if the access operation is a write, and a read miss if the operation is a read. No memory accesses are required for a hit. For a miss, one LOAD is required to fetch data from the memory location into a register if the operation is a read. No fetch is needed for a write. If the register being displaced is dirty, it may need one STORE to update its memory location (if the replaced variable will be read again).

Let us define two look-ahead functions. NEXT(i,x) returns the number of steps from step i to the first step after i which reads symbolic register x. NEXTS(i,x) returns the number of steps from step i to the first step after i which writes x. If no instance of the appropriate access is found, the functions return ∞.

The cost function is defined as follows:

Suppose that Q_{i-1} differs from Q_i in that q_{i-1}^j = (x,s) whereas q_i^j = (x', s') and x $\neq$ x'

1) Update cost:

If s = s_1 cost = 0
If s = s_2 and
(NEXT (i-1,x) = ∞ or
NEXTS(i-1, x) < NEXT(i-1,x))
then cost = 0
else cost = 1

2) Fetch cost:

If s' = s_1 cost = 1
If s' = s_2 cost = 0

C(Q_{i-1}, Q_i) = update cost + fetch cost

The above cost function differs from that of Horwitz in two aspects:

(1) In our model, a write miss does not need an additional (read) memory access. In their model, because the basic operation for writing an index register is a modification, it does. (Probably because the basic operations to write an index register are increment and decrement, Horwitz defines it as a modification) Since modification is a read followed by a write, a modification miss requires a LOAD to fetch data into a register first.

(2) In our model, when a dirty variable is replaced, it is not necessary to update the memory location if the next access of that variable is a write, so a store can then be saved. This never happens in Horwitz's model because there is no write operations, so a modification must read the same data item first.

With our model, index register allocation [Horwitz66, Kennedy72] is just a restricted version of general purpose register allocation.

Now we define the cost function of an allocation as:

$$\text{Cost}(A) = \sum_{i=2}^{i=n} C(Q_{i-1}, Q_i)$$

The goal of this algorithm is to find an allocation A which has minimal Cost(A). The obvious optimal approach to finding such an allocation is to try all possible (legal) allocations for a given program, and to pick the one with least cost. Since the number of legal allocations is finite, this method will produce an optimal allocation. However, since there may be N possible candidates to choose at every step, computation time may grow exponentially. Although the exponentially growing tree may be bounded (since the number of different configurations is finite for fixed N and M), the number of different configurations also grows exponentially proportional to M and N. Table 1 shows the (large) number of different configurations.

Number of Variables	Number of Registers	Number of Configurations
10	2	36
10	4	1344
10	8	9216
15	2	56
15	4	5824
15	8	878592
20	2	76
20	4	15504
20	8	12899328
20	16	254017536
25	2	96
25	4	32384
25	8	88602624
25	16	85688582144
30	2	116
30	4	58464
30	8	399559680
30	16	5082890895360

Table 4.1

In a legal allocation, the symbolic register at the *i*th step must be in the *i*th configuration. The number of different configurations, therefore, is C(M-1, N-1)× 2^N. From Table 4.1, which was generated from this formula, we see that the exhaustive search method is computationally infeasible even for moderate values of M and N. Horwitz *et al* [Horwitz66] provided some rules to restrict this exponential growth; we shall do the same.

4.3. The Rules

We partition a configuration Q_i into four disjoint sets as follows:

set 1 = { x or x^* | x in Q_i, NEXT(i,x) = NEXTS(i,x) = ∞ } This set consists of all the variables which are in Q_i and will never be accessed after step i.

set 2 = { x or x^* | x in Q_i, NEXTS(i,x) < NEXT(i,x) } this set consists of all the variables which will be written after step i before being read.

set 3 = { x | x in Q_i, NEXT(i,x) < NEXTS(i,x) } this set consists of all the variables which are clean and will be read after step i before being written.

set 4 = { x^* | x in Q_i, NEXT(i,x) < NEXTS(i,x) } this set consists of all the variables which are dirty and will be read after step i before being written.

The fourth set will be divided into two subsets, which are:

set 4.1 = { x^* | x in Q_i, NEXT(i,x) < NEXTS(i,x) and x not in (I_j, j = i + NEXT(i,x), i + NEXTS(i,x)) } This is a set of pairs such as x^* --- x ---x^* or x^* --- x --- ∞.

set 4.2 = { x^* | x in Q_i, NEXT(i,x) < NEXTS(i,x) and NEXT(i+ NEXT(i,x), x) < NEXTS(i+ NEXT(i,x), x) } This is a set of pairs such as x^* --- x --- x .

We may now state the rules for the generation of configurations for step i+1 from a configuration Q_i when I_{i+1} is not in Q_i.

Rule 1 :

If there exists an x in set 1, then generate only one configuration Q_{i+1}, in which x is replaced by I_{i+1}.

Else

Rule 2 :

If there exists an x in set 2, then generate only one configuration Q_{i+1}, in which x is replaced by I_{i+1}.

No stores are needed to be issued for x^* in both Rule 1 and Rule 2.

If neither of the above two conditions holds, then

Let C = max{ NEXT(i, x), x in Q_i }, D = max { NEXT(i,y), y^* in Q_i } and x_c be such that NEXT(i,x_c) = C.

Rule 3:

If C > D then generate only one configuration Q_{i+1}, in which x_c is replaced by I_{i+1}.

If rule 3 fails, then

Let C' = max{ NEXT(i,x), x $\in$ set 4.1}, D' = max{ NEXT(i,x), x $\in$ 4.2}, and x_c be such that NEXT(i, x_c) = C'.

Rule 4:

If C' > D' then generate a configuration Q_{i+1}, in which x_c is replaced by I_{i+1} .

For each x^* in Q_i such that x $\in$ set 4.2 and NEXT(i, x) > C, generate a configuration in which x^* is replaced by I_{i+1}.

Suppose we start with an initial configuration Q_0. Let us assign it a cost of 0. Now with each configuration we consider at step i, we associate a cost and a parent pointer. The parent pointer points to the configuration at step i-1 from which the configuration we are presently considering was

generated. The cost of a configuration Q_i is defined as:

$$C(Q_i) = C(\text{parent}(Q_i)) + C(\text{parent}(Q_i), Q_i)$$

Clearly, $C(A) = C(Q_n)$.

4.3.1. If a variable is never used again after step i, as one in the set 1, the displacement cost is 0. This is because the update cost is 0 and there is no future fetch cost.

4.3.2. If a variable x is in set 2, the displacement cost is also 0. Since the update cost is 0 and the coming operation on x is a write, there is no fetch cost.

There is a difference between replacing an element in set 1 and replacing one in set 2. Replacing elements in set 2 will produce misses in the future, while replacing elements in set 1 won't. However, in terms of cost, replacing elements in set 1 is equivalent to replacing elements in set 2. This can be shown in the following way:

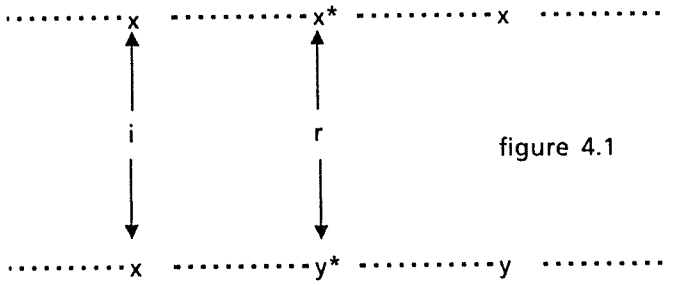

figure 4.1

figure 4.2

Suppose x is in Q_{i-1}, and x $\in$ set 2 (*i.e.*, there is an x^* at step r, r > i, and no x in step i to step r). Now let us rename all the accesses of x after step i to y; then x in Q_{i-1} $\in$ set 2 in figure 4.1, but x $\in$ set 1 in figure 4.2. At step r, if x is not in Q_{r-1}, both figure 4.1 and 4.2 have a write miss, and they should have the same replacement cost. If x is in Q_{r-1}, then figure 4.1 has a hit, while figure 4.2 has a write miss. Nevertheless, in figure 4.2, y can displace x with cost 0. Therefore, variables in set 2 are equivalent to variables in set 1 in terms of cost.

4.3.3. Among elements in set 3, replace the most distant one giving minimal cost. This is Belady's well-known MIN algorithm, commonly used for comparison in replacing pages in a virtual memory system. MIN minimizes the number of misses, which is different from our goal, which is to minimize the number of LOADs and STOREs. Since the elements in set 3 are all clean, there is no update cost. Therefore, minimizing misses is identical to minimizing LOADs. A formal proof of this identity can be found in [Horwitz66]. Here, we only illustrate this property.

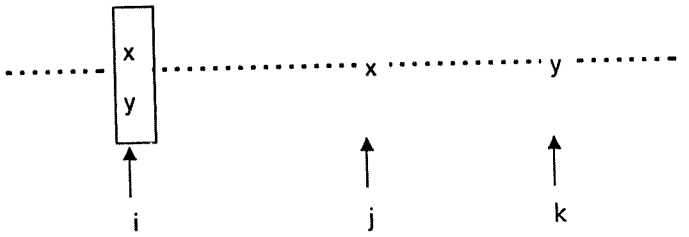

figure 4.3

Suppose x, y in Q_i, I_j = x, I_k = y, such that i < j < k, and that there is no x $\in$ { I_n, n=i+1, j-1 }, and no y $\in$ { I_n, n=i+1,k-1}. Assume I_{i+1} is not in Q_i, and one of x, y will be replaced. Two possible configurations, n_1 and n_2, are produced at step i, and they differ in exactly one element, x $\in$ n_1 and y $\in$ n_2. Let us isolate the remaining N-1 elements from x and y so that they may use the same replacement strategy. At step j, there are two configurations: Q_{j-1}^1, which originates from n_1, and Q_{j-1}^2 which originates from n_2. The cost associated with Q_{j-1}^1 and Q_{j-1}^2 are the same, because they use the same replacement strategy from step i to step j. If

(a) x is in Q_{j-1}^1, this is a hit: $C(Q_j^1) = C(Q_{j-1}^1)$. Because x is not in Q_{j-1}^2, $C(Q_j^2) = C(Q_{j-1}^2) + 1 + U$, where U is the update cost of replacing one element in Q_{j-1}^2. If the replaced element is y, then U is 0 and Q_j^2 has exactly the same configuration as Q_j^1, and replacing y at step i is better. If the replaced element is z and z $\neq$ y then Q_j^1 differs from Q_j^2 in exactly one element, which is z in Q_j^1 and y in Q_j^2. If there is an optimal cost path through Q_j^2, then there is also an optimal path through Q_j^1, because $C(Q_j^1, Q_j^2) = 1 + U$, U is the update cost of z (either 0 or 1), and $C(Q_j^1) + 1 + U = C(Q_j^2)$. Therefore,

replacing y at step i is not worse than replacing x.

(b) x is not in Q_{j-1}^1. Then both Q_{j-1}^1 and Q_{j-1}^2 have a miss at step j, Q_j^1 and Q_j^2 have the same cost.

From the above, it is clearly better to displace the variable with the longer distance. Thus we displace the one which is read farthest away.

4.3.4. If x^*, y in Q_i such that NEXT(i,x^*) < NEXT(i,y), then it is not necessary to generate a configuration which displaces x^*. The proof is similar to that in 4.3.3.

4.3.5. If x $\in$ set 4.1, y $\in$ set 4, and NEXT(i,x) < NEXT(i,y), then it is not necessary to generate a configuration which displaces x. The proof is also similar to that in 4.3.3.

4.4. The Algorithm

The algorithm for generating optimal allocation is described as follows:

A configuration, along with its cost and parent information, is called a node. We start from an initial configuration Q_0 with cost 0, and generate a set of nodes associated with step 1. Note that Q_0 may not necessarily consist of all empty registers. The set of nodes for step i will be called NODESET(i). Subsequent steps are as follows:

1). i := 1, generate the NODESET(1) from Q_0

2). Generate NODESET(i+1).

```
for each node N in NODESET(i) do
      if HIT then N' = N, insert N' into NODESET(i+1), parent(N')=N.
      if MISS then generate nodes from N as described in rules 1, 2,
      3, and 4, above.
od
```

Every time a new node is inserted into NODESET(i), compare it with each node in the set. If two nodes have the same configuration then keep only the one with the smaller cost.

3). If i < n, i := i+1 goto 2.

4). Find the node N_{min} in NODESET(n) with the least cost. Trace back through the parent pointers to get the allocation.

4.5. Examples

Example 1:

Assume there are only two real registers. The input program is:

$r_1^*\ r_2\ r_3\ r_2^*\ r_1$

The replacement process is shown in the following table.

	sequence	r1*	r2	r3	r2*	r1	Results
Belady's	miss	x	x	x	-	x	4
	replace	-	-	r1	-	r3	-
	load	0	1	1	0	1	3
	store	0	0	1	0	0	1
	configu-ration	- -	r1* -	r1* r2	r3 r2	r3 r2*	r1 r2*
Ours	miss	x	x	x	x	-	4
	replace	-	-	r2	r3	-	-
	load	0	1	1	0	0	2
	store	0	0	0	0	0	0
	configu-ration	- -	r1* -	r1* r2	r1* r3	r1* r2*	r1* r2*

Table 4.2

Belady's MIN algorithm produces 4 misses; it needs 3 LOADs and 1 STORE. The above algorithm also produces 4 misses, but it needs only 2 LOADs and no STOREs. Belady's MIN algorithm can minimize number of LOADs only if the write through policy is used. From the above example, we understand that Belady's MIN algorithm can not minimize the number of LOADs if the

write back policy [SmithA82] is used instead.

Example 2:

Assume there are two real registers, and the input program is:

$r_1^* \; r_2^* \; r_3 \; r_2 \; r_1 \; r_4^* \; r_5^* \; r_4 \; r_5 \; r_2 \; r_1^* \; \ldots$

The replacement process is shown as follows:

		r1*	r2*	r3	r2	r1	r4*	r5*	r4	r5	r2	r1*	Results
A	miss	x	x	x	-	x	x	x	-	-	x	x	8
	replace	-	-	r1	-	r3	r1	r2	-	-	r4	r5	-
	load	0	0	1	0	1	0	0	0	0	1	0	3
	store	0	0	1	0	0	0	1	0	0	0	0	2
	configu-ration	- -	r1* -	r1* r2*	r3 r2*	r3 r2*	r1 r2*	r4* r2*	r4* r5*	r4* r5*	r4* r5*	r2 r5*	r2 r1*
B	miss	x	x	x	x	-	x	x	-	-	x	x	8
	replace	-	-	r2	r3	-	r1	r2	-	-	r4	r5	-
	load	0	0	1	1	0	0	0	0	0	1	0	3
	store	0	0	1	0	0	0	0	0	0	0	0	1
	configu-ration	- -	r1* -	r1* r2*	r1* r3	r1* r2	r1* r2	r4* r2	r4* r5*	r4* r5*	r4* r5*	r2 r5*	r2 r1*

Table 4.3

At step 3 there is a miss. If we displace r_1 (as in A), eventually the least cost will be 5. If we displace r_2 instead (as in B), then a minimal cost of 4 can be obtained. Note that r_1 is farther away than r_2, but displacing r_1 produces a higher cost than displacing r_2. This example is designed to clarify the point that, for elements in set 4, access distance is not the only factor to consider when making a replacement decision: No rigid order relation exists as it does in set 3.

4.6. Existing Algorithms

Freiburghouse evaluated four allocation algorithms in [Freib74] - Belady's, Usage Count, LRU, and LRL (Least-Recently-Loaded). These four algorithms assume that values in registers are always consistent with their corresponding values in memory, and consequently do not need to be stored when displaced. This "write through" assumption decreases performance, and should be relaxed, since it uses more STOREs than necessary. The approach presented above is similar to a "write back" strategy. Furthermore, dead registers, for which the next access is a write, require no update. Kim and Tan [Kim78] have extensively studied register spilling problems. Their "life range analysis" is similar to ours. When a register must be displaced, they pick the most distant one; I call it the Farthest First (FF) algorithm. Fischer presents an algorithm [Fischer82] which also tries to eliminate unnecessary stores. When spilling is needed, his algorithm chooses the most distant clean variable first; if there is no clean one left, the most distant dirty one is displaced. We call this the Clean First (CF) algorithm. Aho and Ullman [Aho77] gave no suggestion with respect to clean or distant variables, in their heuristic algorithm.

When replacement is necessary, how to choose the best one to displace ? If the most distant one is clean then no other need be considered. If the most distant is dirty, however, a decision must be made between the most distant clean variable and those dirty ones which have distances longer than it. On the one hand, replacing the most distant one minimizes the number of misses which, in turn, often reduces LOADs and STOREs. Choosing a clean variable as victim, on the other hand, may reduce the number of STOREs by one. The algorithm described above chooses the best one, but it requires exponential time in the worst case. A heuristic algorithm which makes a compromise between Fischer's CF algorithm and Kim's FF algorithm is described below.

4.7. A Heuristic Algorithm

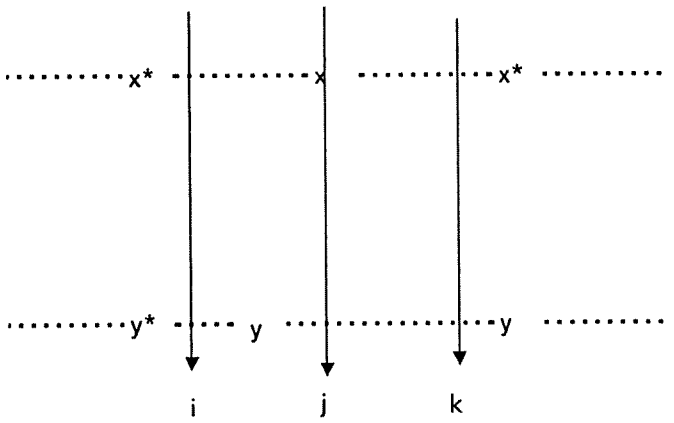

figure 4.4

From example 2 in the section 4.5, we have learned that spilling a y^*--y---y^* might be better than spilling a x^*--x---x, even when the distance x^*--x is longer than the distance y^*--y (see figure 4.4). This is because spilling x at step i needs one STORE, but spilling it at step k needs no STOREs. But one STORE is required to spill y, regardless of whether it is spilled at step i or step k. Therefore, if x remains in the register until step j, one STORE might be saved. This save will occur only when the distance of y^*--y is close to that of x^*--x. In other cases, additional misses may require additional LOADs and STOREs.

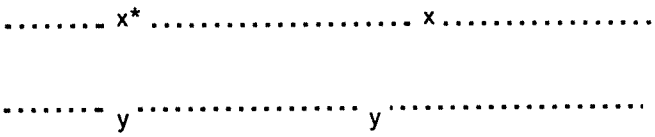

figure 4.5

From figure 4.5 it is clear that if the distance of y--y is close enough to that of x^*--x, then replacing y may be cheaper. The two alternatives may involve the same number of misses, but replacing y needs no STOREs. This, again, is a trade off between saving STOREs and reducing misses. The approach that we propose is to use a weighted distance as the selection criterion. For clean pairs (like y--y) the distance is multiplied by a weight w_1, $w_1 > 1$; For x^*--x--x^* pairs, a positive $w_2 < 1$ is assigned. If the w_1 is equal to 1, this approach is the same as the FF algorithm; if the w_1 is equal to a very large number it "degenerates" into the CF algorithm. Thus, the weighted algorithm will always have a result in intermediate between that of FF and that of CF. With a properly selected w_1, its result will be close to the lower of the two (FF and CF). In addition, with the

help of w_2, it may outperform both FF and CF. The optimal values of w_1 and w_2 depend on input programs and the number of available registers. We are currently investigating a systematic method of finding an appropriate weight for w_1 and w_2 adaptively. For example, increasing w_1 when the miss ratio is getting lower and decreasing w_1 when the miss ratio is getting higher. This is motivated by the belief that, when the miss ratio is high, decreasing misses is more important than saving stores. Experimentally, assigning w_1 in the range of 1.2 to 1.5 produces good results.

The heuristic algorithm is as follows:

1) i := 1, generate Q_1 from Q_0.
2) while i < n do
   i := i+1
   Generate Q_i from Q_{i-1} as follows :
   2.1) If there is a HIT at step i then $Q_i := Q_{i-1}$.
   2.2) If there is a MISS then
      2.2.1) if there is a x, x in set 1 or set 2 then
         replace x by I_i.
      2.2.2) max := 0
         r := 0
       foreach x_j in Q_{i-1} do
         case
            $x_j \in$ set 3 : distance := NEXT(i-1,x_j) * w_1
            $x_j \in$ set 4.1 : distance := NEXT(i-1, x_j) * w_2
            $x_j \in$ set 4.2 : distance := NEXT(i-1, x_j)
         esac
         if distance > max then
         max := distance, r := j
       od
       replacing x_r by I_i.
od

4.8. Simulation Results

To evaluate the effectiveness of different heuristic algorithms and compare them to the optimal one, we used a trace-driven simulation. We compiled C programs and traced the assembly language program, and converted a sequence of instructions into a sequence of register reads and writes. Then we used the sequence of register reads and writes as input data. To get more symbolic registers, we assign as many local variables to registers as possible. Since our C compiler takes up to 6 register hint variables, we have to allocate more variables to registers at the assembly level. Usually

the basic block we can get from the assembly program is small; we intentionally trace some control flow to get a larger block, as with trace scheduling [Fisher81]. Further, in order to get more local variables (so that more symbolic registers will be used), we expanded some procedure calls in-line. The following programs were used.

1 A basic block from unix utility *grep*.
2 A basic block from unix utility *sort*.
3 A basic block from a subroutine in a cache simulator program.
4 An enlarged basic block (by trace scheduling) from a cache simulator.
5 An enlarged basic block (by in-line expansion) from a cache simulator.
6,7 are artificially derived program segments.
8 A artificially derived program with in-line expansion and trace scheduling to get larger basic block.

Number of LOADs and STOREs Generated							
Input Trace	# of regs	LRU	Belady's	FF	CF	Mixed w1 = 1.25 w2 = 0.98	True Optimal
1	2	34	24	19	20	19	18
	4	13	9	4	4	4	4
	8	3	3	3	3	3	3
2	2	27	20	14	14	14	14
	4	15	9	6	6	6	6
	8	7	6	6	6	6	6
3	2	28	22	14	12	12	12
	4	19	13	6	6	6	6
	8	10	7	5	5	5	5
4	2	47	38	24	24	24	24
	4	32	24	13	13	13	13
	8	16	16	11	11	11	11
5	2	83	70	46	43	43	43
	4	62	49	21	25	22	21
	8	47	39	13	13	13	13
6	2	35	28	18	20	19	18
	4	28	20	9	10	9	9
7	2	42	38	29	28	28	27
	4	39	26	12	12	12	11
8	2	110	89	75	74	73	71
	4	85	61	34	37	34	34
	8	51	42	17	16	16	15

Table 4.4

These simulations showed that for small basic blocks with small numbers of variables (1,2,3), the three algorithms (FF, CF, and Mixed) have performance identical to that of the optimal

algorithm. This means, in realistic cases, a heuristic algorithm like FF or CF is good enough. For larger basic blocks with more variables (5,8), the heuristic algorithms produce slightly more LOADs and STOREs than the optimal one. Among the heuristic algorithms, each of FF and CF wins in some cases, while our algorithm has intermediate results close to the lower. We believe our algorithm is more appropriate than the other two for large basic blocks with many variables. In the next chapter, we discuss how to create larger basic blocks.

Using future access information, we can save unnecessary STOREs, thus reducing the bus traffic to a large extent. In table 4.4, the performance difference between the LRU method and any one of the three heuristic algorithms is significant. This suggests that registers may significantly outperform cache memories in a VLSI processor.

Relative Computation Time (seconds)			
Input Trace	# of regs	Mixed	True Optimal
4	2	0.2	0.8
	4	0.2	0.5
	8	0.2	0.5
5	2	0.2	2.2
	4	0.2	4.6
	8	0.2	1.0
7	2	0.1	0.8
	4	0.2	139.7
8	2	0.3	625.5
	4	0.3	2306.9
	8	0.3	301.6
	16	0.4	0.8

Table 4.5

The optimal algorithm usually requires reasonable computation time, but it occasionally does not (trace 8 with 4 registers). The heuristic algorithm requires less computation time than does the optimal one and is highly predictable.

4.9. Comparison with graph coloring algorithm

If the interference graph of a straight-line program is N-colorable, the graph coloring algorithm will give an allocation with no additional LOADs and STOREs, that is, an allocation with cost 0. If a graph is N-colorable then, at any program point, the number of live registers will not exceed N. This implies that there exists at least one empty register or dead variable that can be replaced at no cost. Thus, an allocation with cost 0 can also be found by our algorithm. When the interference graph is *not* N-colorable, the graph algorithm blocks and has to resort to a replacement algorithm. The spilling algorithm used by [Chaitin82] is similar to a reference count algorithm [Aho77,Powell84], which does not adequately consider clustered accesses. We therefore believe that the replacement based algorithm is superior to the graph coloring algorithm in straight-line programs.

Another drawback of the graph coloring algorithm is that it may introduce accidental constraints to code scheduling [Auslander82]. The PL.8 compiler solves this by doing code scheduling both before and after register allocation. The problem with this solution, however, is that if the original program is not N-colorable, then code scheduling will make allocation even harder, since code scheduling tends to lengthen register lifetime and cause more interferences. If the replacement-based allocation algorithm is used, however, then by consulting the data dependence graph simultaneously, fewer accidental constraints are introduced. We address the conflict between register allocation and code scheduling in greater detail in a later chapter.

4.10. Summary

We have described a model and derived an optimal register allocation algorithm for a straight-line program. A heuristic algorithm which has results close to this optimal one is also described. The replacement-based allocation algorithm performs both allocation and spilling during the same pass rather than using multiple passes (as in the graph coloring algorithm). If the algorithm can only be used in straight-line programs, however, it appears to have limited value, since few programs are branch-free. (In fact, basic blocks are usually small and, therefore, numerous [Patterson85,

Fisher81].) We discuss how to extend this algorithm to global allocation in the next chapter.

Both Belady's and Horwitz's algorithms may be used to evaluate performance of a paging system. Belady's algorithm estimates the upper bound of page fault ratio, while Horwitz's algorithm measures the least number of disk I/Os needed. Our algorithm is not adequate for a paging system: In our model, the write unit is equal to the transfer unit, so some STOREs can be saved. In a paging system, however, the write unit (normally a word) is much smaller than the transfer unit (a page), so the chance to save these "unnecessary" stores is slim.

5. Global Allocation

In the previous chapter, we discussed local, or intra-basic-block register allocation. Now we want to extend our algorithm to global, or inter-basic-block register allocation. The approach we used in local allocation used registers to hold values for the duration of a single basic block. However, when this approach is used globally, we were forced to store values of the variables which are dirty and live on exit at the end of each block. To save some of the stores and corresponding loads, we must keep these registers consistent across block boundaries.

Global register allocation is a very difficult problem. Practical implementations are necessarily heuristic and typically complex [Allen81, Auslander82, Chow83]. Since programs spend most of their time in inner loops, most of the global allocation algorithms pay much attention to allocating registers for variables in inner loops.

5.1. Previous Work

Lowry and Medlock [Lowry69] used a simple algorithm in the FORTRAN H compiler. They identify which variables, constants, and base addresses are referenced most frequently (statically) within a loop and assign many of them to registers across the loop. This simple algorithm is very effective for small loops.

Day [Day70] formulates global register allocation as an integer programming problem. He divides the allocation problems into three different kinds -- global one-one, many-one, and many-

few. Branch and bound algorithms which lead to optimal solutions are given for many-one and many-few problems, while heuristic algorithms are given for all three problems. The algorithm for one-one allocation is actually the same one as used in the FORTRAN H compiler with improvements. Interferences among variables are analyzed so that more than one data item can be allocated in one register. His algorithms are based on the IBM/360 model, for which data items can be accessed directly from storage. Our model is based on a LOAD/STORE architecture for which data items must be loaded into registers before being used.

Beatty [Beatty74] separates the register assignment process into three steps-- local allocation, global assignment and local assignment. He starts with locally allocated variables, and extends their lifetimes by moving their loads and stores to less frequently executed parts of the program. His algorithm is more flexible than the approach used in the FORTRAN H compiler which cannot eliminate individual loads and stores of variables which have not been allocated in registers.

Harrison [Harrison75] applies Belady's MIN algorithm to the flow constructs of real programs. Global flow analysis techniques are used to gather information which guides the register allocation. Branch frequency information is also used so that variables in the most frequently executed part will be allocated in registers first. In addition, Harrison discusses several extensions to his algorithm, like handling special register characteristics and doing code selection at the same time as register allocation.

Wulf, *et al.* [Wulf75] separate register allocation from register assignment in their Bliss-11 compiler. They assign temporary names (TN) to selected variables, common subexpressions, and loop invariants in the TN phase and map them to real registers in the PACK phase. The order in which TN's are considered for assignment to registers depends on an accumulation and balancing of costs. TN's that occur in loops are given more weight than others. Lifetimes of TN are also analyzed so that two TN's can share a register if they do not interfere with each other.

Kim [Kim78, Kim79], like Beatty, describes a system for manipulation of individual loads and stores. First, he assumes that all variables are allocated in registers. Then, he uses lifetime analysis

to determine sections of the program over which the number of variables that are alive exceeds the number of available registers, so that some of them must be spilled to memory. Instructions to store and load those spilled variables are inserted into the program, and moved around to the so-called "edge block". The corresponding loads and stores in the "edge block" can be removed. Also, loads and stores can be moved to less frequently executed parts of the program.

Chaitin, *et al.*, [Chaitin81, Chaitin82] use graph coloring techniques to do global register allocation in the PL.8 optimizing compiler. Though the graph coloring technique has been suggested by Yershov [Yershov71], Cocke [Allen76], and others, it is successfully developed and implemented by Chaitin. The concept of graph coloring technique has been described briefly in chapter 2. When the compiler cannot color the register conflict graph, it must add code to spill some nodes. Spill decisions are made on the basis of the register conflict graph and cost estimates of the value of keeping the variable in a register rather than in memory. The cost of spilling a node is approximately equal to the number of references to that variable, where each reference is weighted by its estimated execution frequency. Chaitin et al., assume that each instruction in a loop is executed ten more times than it would be if it were outside the loop. The graph coloring model is uniform and systematic in its handling of machine idiosyncrasies.

Chow and Hennessy [Chow84] also use the graph coloring techniques for global register allocation in their UOPT, a portable machine-independent global optimizer. They adopt the notion of priorities in node-coloring. The assignment of priorities is based on estimates of the benefits that can be derived from allocating variables in registers, including allowances for loop nesting depth and access variable clustering. Each node in the interference graph is a live range for some variable. If the number of colors are not enough for coloring the interference graph, the compiler will assign colors to those nodes which have higher saving cost. Then by splitting long live ranges into short subranges, a variable may be assigned to a register for a short time. Chow and Hennessy's model is a little different from Chaitin's, since they assume each variable can be accessed from storage directly. The coloring process is therefore easier to terminate. (Remember that in Chaitin's model ,

the spilling process must be iterated until the graph can be colored. In Chow's model, it is easy to stop the coloring process just by assigning colors to those nodes with higher saving cost and leave the uncolored ones in storage)

5.2. Branch Frequency

A register allocation algorithm could allocate registers more effectively if it could predict which parts of a program will execute more frequently. Ideally, we would like to know not only which loops execute more frequently, but also relative execution frequency for different subsections within a given loop. We know of only one, older compiler which attempted "hot spot" optimization of this sort : the original FORTRAN I compiler, which used branch frequency information in its optimization phase [Backus57]. IBM probably abandoned this technique in later FORTRAN compilers because of the success of the graph coloring algorithm and the implementation difficulties involved in determining branch probabilities.

Traditional static analysis techniques, which assume equal branch probabilities for all program paths, fail to generate good code when one branch is taken most of the time. Consider the example in figure 5.1.

Assume there are only two registers available. Because the program in the figure 5.1 is a loop, the lifetime of each variable is equally long -- from the entry of the loop to the end of the loop. Variables interfere with each other. Two colors are not enough for coloring the interference graph, so spilling is necessary. Both coloring algorithms will spill *A* and *B*, due to their definition of cost. But what if the *true* branch in the above example is taken 90% of the time? Shouldn't we spill variable *C* and *D* instead? The ideal allocation would be as in figure 5.2.

To obtain the allocation in figure 5.2, we first do allocation for the most frequently used path, then for the next frequently used one, .., etc. This principle has been widely used - optimizing more frequently used items while letting the less frequently used ones pay some extra price.

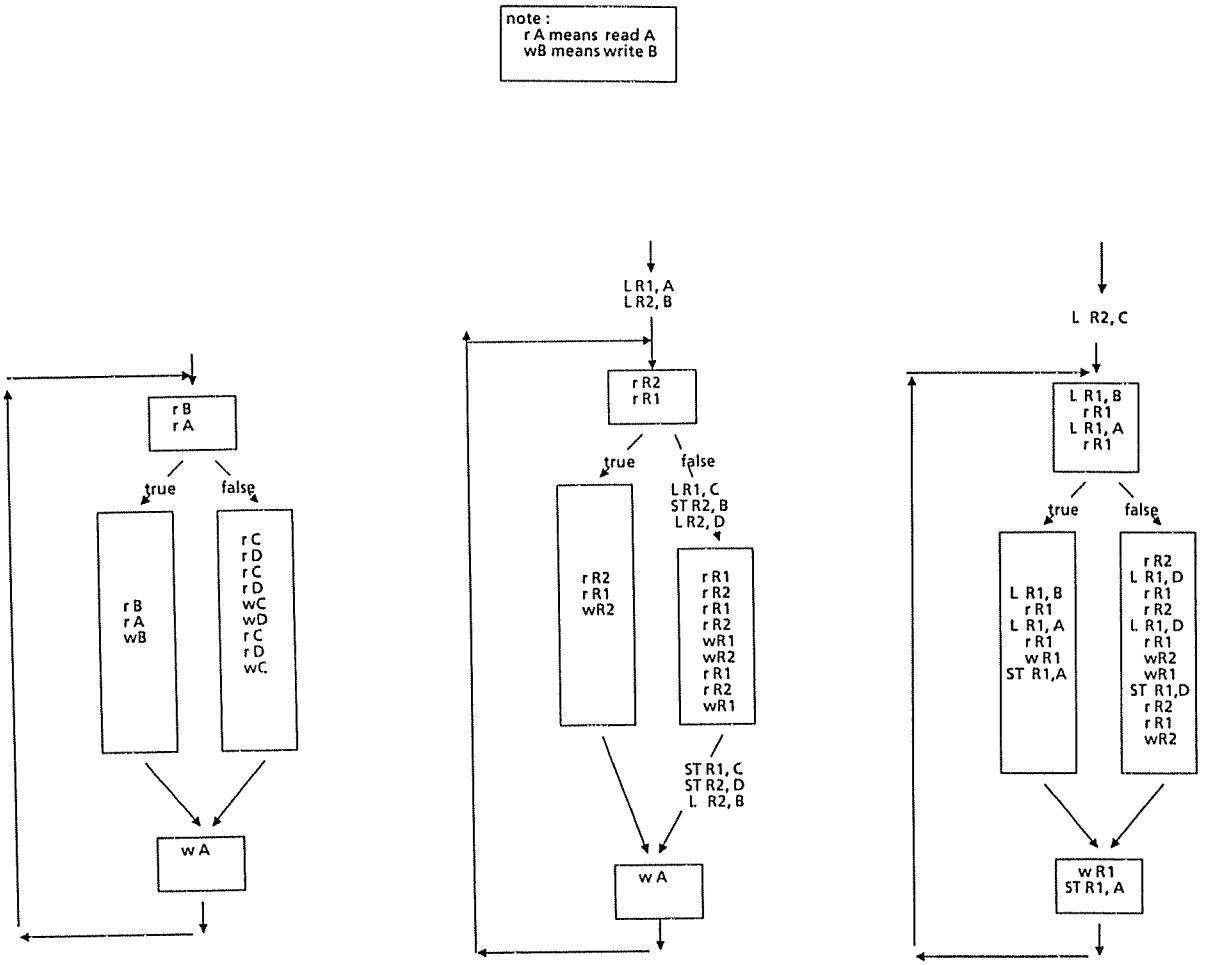

figure 5.1

figure 5.2

figure 5.3

Assume we use the LOAD/STORE architecture for which data items must be loaded into register before being used. A traditional allocation will be as in figure 5.3.

Suppose the loop will be executed 100 times, then the number of LOAD/STORE instructions needed are :

Number of LOAD and STORE Instruction Executed		
Percentage of true branch	Traditional	Ours
95%	601	32
90%	601	62
50%	601	302
30%	601	422
10%	601	542

The FORTRAN I compiler is the first compiler to exploit the idea of optimizing a trace. Recently, Fisher uses Trace Scheduling to do global compaction successfully in the BullDog compiler [Fisher84].

There are three ways to get information of branch probabilities: (1) using an automatic profiler which supplies execution counts based on sample runs of the program; (2) using programmer-supplied hints and loop nesting depth to make reasonable guesses; (3) use symbol analysis of conditional branches; and the second approach is successfully used in the Bulldog compiler.

5.3. Extending the Local Algorithm to Global Allocation

In the previous chapter, we have mentioned that the deficiency of our local allocation algorithm is that basic blocks are usually too small. Common techniques for producing larger basic blocks, including loop unrolling, loop jamming, unswitching, in-line code expansion, avoidance of short circuit branching, and elimination of redundant conditional branches through constant propagation [Allen72], do not increase block size as much as we would like. Selecting a path (or trace), on the other hand, gives us an extremely large block, making it possible to effectively use our algorithm.

There are problems to be worked out when applying our local allocation algorithm to the global allocation.

(1) In local allocation, if the next access of a variable is a write, then no stores are needed to update its value in memory when the variable is spilled. This is not true in global allocation. We have to use global flow analysis to tell whether a variable at a point is dead or not - a

variable is dead at a point if its current value will never be used again. No stores are needed for spilling a dead variable.

(2) Initial allocation needs to be selected carefully if the trace is in a loop. The approach used in the FORTRAN I compiler was as follows : The loop was first considered to be concatenated with a second copy of itself, and allocation carried out in normal fashion through the first of the two copies, with look-ahead extending into the second copy. The contents of registers on exit from the first copy thus determined was now applied as the initial condition.

(3) There are overheads when a less frequently used path rejoins a more frequently used one. Algorithms for minimizing this overhead are under investigation.

(4) The instructions inserted for spilling should be moved to less frequently executed parts of a program (e.g. outside a loop). Kim's work [Kim79] will be useful for solving this problem.

Detailed algorithms are under development. We are also looking at methods to collect run time branch statistics. Implementation of this algorithm will be on the experimental compiler PPC (PIPE PASCAL Compiler)[Young85].

6. Summary

To achieve high performance from a single-chip processor, two problems must be solved: (1) memory latency; (2) limited off-chip bandwidth. To solve these two problems, on-chip local memory, especially registers, must be used effectively.

In a conventional compiler, register allocation is usually done within a procedure. The use of many registers often has the adverse effect of increasing the cost of procedure calls. In order to reduce the overhead of procedure calls, procedure integration techniques, which can remove some of the overhead of procedure calls, have been used. However, procedure integration does not remove the register save/restore overhead entirely. Rather, it shifts the save/restore problem from procedure calls to register allocation. It is necessary to have an efficient algorithm to handling spilling so that the number of LOAD/STORE instructions can be minimized.

In section 3, we examined a "perfect" register allocation scheme based on a LOAD/STORE architecture. The experimental results show that a medium number of registers has the potential for eliminating most of the memory accesses from the *stack*. Therefore, we explored two techniques to obtain a more effective register allocation algorithm. They are: (1) effectively using look-ahead information which is collected by program flow analysis; (2) using branch prediction to perform trace optimization.

In section 4, we developed an optimal algorithm to minimize the number of LOAD/STOREs needed for register allocation in a straight-line program. We also demonstrated that a heuristic algorithm that uses adjustable weights for selecting a victim for spilling produces results close to the optimal one.

Register allocation can be more effective if run-time branch frequency distributions are known. In section 5, we tried to extend our algorithm to global allocation by taking run-time branch frequency into account. The basic approach of our global allocation algorithm is to use the concept of trace optimization.

We are currently studying the implementation details of the global allocation algorithm. This report is concentrating on using registers for the stack, we are also looking for effective ways to handle global data accesses. Vector registers and architectural queues are two possible features for global data accesses. Because they can be used to take the advantage of prefetching without increasing off-chip communication.

7. Acknowledgements

I wish to thank James Goodman for his supervision and supporting the trace facility. I received many useful suggestions from Charles Fischer and Marvin Solomon. I would like to thank P. B. Schechter and P. Pfeiffer for their help in revising this report.

8. References

[Aho72] Aho, Alfred V., and J.D. Ullman, "Optimization of straight line code," SIAM Journal of Computing, 1, 1972.

[Aho77] Aho, Alfred V., and Jeffery D. Ullman, "Principles of Compiler Design", Addison-Wesley, Reading. Mass., 1977.

[Allen72] Allen, F. E., and John Cocke, "A Catalogue of Optimizing Transformations" in Rustin[1972].

[Allen76] Allen, F.E., and J. Cocke, "A Program Data Flow Analysis Procedure" CACM March 1976, Vol 19, No 3.

[Allen80] Allen, F.E., J. L. Carter, J. Fabri, J. Ferrante, W.H. Harrison, P.G. Loewner, and L.H. Trevillyan, "The Experimental Compiling System," IBM J. Res. Develop. 24, 695-715, 1980.

[Allen81] Allen, F.E., "The History of Language Processor Technology in IBM", IBM J. RES. Develop. Vol. 25 Sept. 1981.

[Alpert83] Alpert, D., D. Carberry, M. Yamamura, Y. Chow, and P. Mak, "32-bit Processor Chip Integrates Major System Functions," Electronics, pp. 113-119, 14 July 1983.

[Auslander82] Auslander, Marc and Martin Hopkins, "An Overview of the PL.8 compiler" Proceedings of the SIGPLAN '82 Symposium on Compiler Construction.

[Backus57] Backus, J. W., *et al.*, "The FORTRAN Automatic Coding Sytem," 1957, in [Rosen67].

[Ball79] Ball, J. Eugene "Predicting the Effects of Optimization on a Procedure Body" SIGPLAN'79 Symposium on Compiler Construction, SIGPLAN Notice 1979

[Beatty72] Beatty, J.C., "A Global Register Assignment Algorithm," in Rustin[1972]

[Beatty74] Beatty, J. C., "Register assignment algorithm for generation of highly optimized object code," IBM J. R & D, 1974, 20-40.

[Belady66] Belady, L. A., "A Study of Replacement Algorithms for A Virtual-Storage Computer" IBM Systems Journal, Vol. 5, No. 2, 1966.

[Carter77] Carter, J. L., "A case study of a new code generation technique for compilers," CACM, Vol 20, pp 914-920, 1977.

[Chaitin81] Chaitin, G.J., Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins and Peter W. Markstein, "Register Allocation Via Coloring" Computer Language 6, 1981.

[Chaitin82] Chaitin, G.J., "Register Allocation and Spilling Via Graph Coloring" SIGPLAN 82 Symposium on Compiler Construction.

[Chow83] Chow, Frederick C., "A Portable Machine-Independent Global Optimizer -- Design and Measurements" Ph. D. Dissertation Dec. 1983.

[Chow84] Chow, Frederick, and John Hennessy, "Register Allocation by Priority-Based Coloring " Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.

[Cocke70] Cocke J., and J. T. Schwartz, "Programming Languages and Their Compilers" Courant Institute of Mathematical Sciences, NYU, 1970.

[Cooper84] Cooper, Keith D., "Analyzing Aliasing of Reference Formal Parameters," POPL 1984.

[Dannen79] Dannenberg, R. B., "An Architecture with Many Operand Registers to Efficiently Execute Block-Structured Languages," Proc. of the 6th Annual Symposium on Computer Architecture, April 1979.

[DEC82] DEC,"Three different VAX C Compilers," unpublished internal memorandum, March 1982.

[Day70] Day, W. H. E., "Compiler Assignment of Data Items to Registers," IBM Syst. J. 9, 281-317, 1970.

[Ditzel82] Ditzel, D., and R. McLellan, "Register allocation for free: The C machine stack cache," Symposium on Architecture Support for Programming Languages and Operating Systems, 1982.

[Dongarra79] Dongarra, J.J., and A. R. Jinds, "Unrolling Loops in FORTRAN," Software Practice and Experience 9, 3, pp. 219-226, March 1979.

[Fischer82] Fischer, C. N., "Notes on Advanced Topics in Compiler Construction," Class Notes for the course CS703 at UW-Madison, 1982.

[Fisher81] Fisher, J., "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, Vol. C-30, No. 7, July 1981.

[Fisher84] Fisher, Joseph A., and John R. Ellis, "Parallel Processing: A Smart Compiler and a Dumb Machine," SIGPLAN Symposium on Compiler Construction 1984.

[Freib74] Freiburghouse, R.A., "Register allocation via usage counts," CACM 17:11 638-642 Nov. 1974.

[Goodman83] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," Internatinal Architecture Conference, 1983.

[Goodman85a] Goodman, J. R., and Honesty C. Young "Code Scheduling Methods for Some Architecture Features in PIPE," Computer Sciences Technical Report #579 Feb. 1985.

[Goodman85b] Goodman, J.R., "Cache Memory Optimization to Reduce Processor/Memory Traffic," Computer Science Technical Report #580, Feb. 1985.

[Harrison75] Harrison, William, "A Class of Register Allocation Algorithms," RC 5342, IBM Research Report, 1975.

[Hasegawa85] Hasegawa, Makoto and Yoshiharu Shigei, "High-Speed Top-Of-Stack Scheme for VLSI Processors: a Management Algorithm and Its Analysis," Proceeding of the 12th Annual International Symposium on Computer Architecture, June 1985.

[Hennessy81] Hennessy, John, Norman Jouppi, Forest Baskett, and John Gill, "MIPS: A VLSI Processor Architecture," Technical Report No. 223, Computer Systems Laboratory, Standford University, Nov. 1981.

[Hennessy83] Hennessy, John, and Thomas Gross, "Postpass Code Optimization of Pipeline Constrains," ACM Transactions on Programming Languages and Systems 5, 3, pp. 422-448, July 1983.

[Hennessy84] Hennessy, John L., "VLSI Processor Architecure," IEEE Transactions on Computers Vol. c-33 No. 12, Dec. 1984.

[Hill84] Hill, Mark D., and Alan Jay Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories," Proc. Elevnth Annual International Symposium on Computer Architecture, June 1984.

[Hitchcock85] Hitchcock III, Charles Yand, and H. M. Brinkley Sprunt "Analyzing Multiple Register Sets," The 12th Annual International Symposium on Computer Architecture, 1985.

[Horwitz66] Horwitz L.P., Karp R. M., Miller R. E., and Winograd S. "Index Register Allocation". J. ACM 13, 1, Jan. 1966, 43-61.

[Kennedy72] Kennedy, Ken, "Index Register Allocation in Straight Line Code and Simple Loops," in Rustin[1972].

[Kim78] Kim, J., "Spill Placement Optimization in Register Allocation for Compilers," RC 7251, IBM Research Report, 1978.

[Kim79] Kim, J., and C. J. Tan, "Register Assignment Algorithms For Optimizing Micro-Code Compilers-- Part I," RC 7639, IBM Research Report, 1979.

[Ledgard81] Ledgard, H., "Ada: An Introduction," Springer Verlag, 1981.

[Lee84] Lee, Johnny K.F., and Alan Jay Smith, "Branch Prediction Strategies and Branch Target Buffer Design," Computer, Vol. 17, Jan. 1984.

[Leverett81] Leverett, Bruce W., "Register Allocation in Optimizing Compilers," Ph.D. Thesis, Feb. 1981.

[Lowry69] Lowry, E.S., and C.W. Medlock, "Object Code Optimization," CACM 20, 13-22, 1969.

[Luccio67] Luccio, F., "A Comment on Index Register Allocation," CACM, Vol 10, Number 9, 1967, 572-574.

[Maclaren84] MacLaren, M. Donald "Inline Routines in VAXELN Pascal" ACM SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notice Vol. 19, No. 6, June 1984.

[Madhavji82] Madhavji, N. H. and I. R. Wilson, "CRAY Pascal," Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, 1982

[Moto82] Motorola, "The MC68020 Enhanced M68000 Microprocessor," Product Review,

Motorola Semiconductors, Austin, Texas, March 1982.
[Myers81] Myers, Glenford J., "Advances in Computer Architecture," second edition, Intel Corporation, Santa Clara, California, 1981.
[Patterson80] Patterson, David A., and Carlo H. Sequin, "Design Considerations for Single-Chip Computers of the Future," IEEE Transactions on Computers, Vol. C-29, No. 2, Feb. 1980.
[Patterson82] Patterson, David A., and Carlo H. Sequin, "A VLSI RISC," IEEE Computer, 15, 9, pp. 8-21, Sept. 1982.
[Patterson83] Patterson, David, A., P. Garrison, M.D. Hill, D. Lioupis, C. Nyberg, T.N. Sipple, and K.S. Van Dyke, "Architecture of a VLSI Instruction Cache for a RISC," Proc. Tenth International Symposium on Computer Architecture ," pp. 108-116, June 1983.
[Patterson85] Patterson, David A, "Reduced Instruction Set Computers," CACM Jan. 1985.
[Perkins79] Perkins, Daniel R., and Richard L. Sites, "Machine-Independent Pascal Code Optimization," SIGPLAN Symposium on Compiler Construction 1979.
[Popek77] Popek, J., Horning, J., Lampson, B., *et al*, "Notes on the design of Euclid," SIGPLAN Notices, Vol 12, No. 3, March 1977.
[Powell84] Powell, Michael L., "A portable optimizing compiler for Modula-2," Proceedings of the SIGPLAN Compiler Construction Conference, June, 1984
[Radin82] Radin, G., "The 801 Minicomputer", Symp. on Architecture Support for Programming Languages and Opterating Systems, March 1982, pp. 39-47.
[Ragan83] Ragan-Kelly, R., "Performance of the pyramid computer," Proc. COMPCON, Feb. 1983.
[Rosen67] Rosen, S., "Programming System and Languages," McGraw-Hill, N.Y., 1967.
[Rustin72] Rustin, R., "Design and Optimization of Compilers," Prentice-Hall, Englewood Cliffs, N.J., 1972.
[Schaefer79] Schaefer, M., "A Mathematical Theory of Global Program Optimization," Prentice-Hall, Englewood Cliffs, N.J., 1979.
[Scheifler77] Scheifler, Robert W., "An Analysis of Inline Substitution for a Structured Programming Language," CACM, 20, 9, Sept. 1977, 647-654.
[Sethi70] Sethi, R., and J. D. Ullman, "The Generation of Optimal Code for Arithmatic Expressions," JACM, 17, October, 1970.
[Sethi75] Sethi, R., "Complete Register Allocation Problems," SIAM J. Computing 4:3, 226-248. 1975.
[Sherburne82] Sherburne, Robert W., Jr. Manolis G.H. Katevenis, David A. Patterson, and Carlo H. Sequin, "Datapath Design For RISC," 1982 Conference on Advanced Research In VLSI, M.I.T.
[Sherburne83] Sherburne, R., M. Katevenis, D. Patterson, and C. Sequin, "Local Memory in RISCs," in Proc. Int. Conf. Computer Design, IEEE, Rye, NY, 1983.
[Sites79] Sites, Richard L., "Machine-independent Register Allocation" SIGPLAN Symposium on Compiler Construction 1979.
[Sites79a] Sites, R. L., "How to use 1000 Registers," Proc. of the Caltech Conference on VLSI, Jan. 1979.
[SmithA82] Smith, Alan Jay, "Cache Memories," ACM computing surveys 14, 3, pp 473-530, Sept. 1982.
[SmithA85] Smith, Alan Jay, "Sprunt Cache Evaluation and the Impact of Workload Choice," IEEE/ACM the 12th Annual International Symposium on Computer Architecture, June, 1985.
[SmithJ81] Smith, James E., "A Study of Branch Prediction Strategies," Proceedings, IEEE/ACM the Eighth Annual International Symposium on Computer Architectures, pp. 135-142, May 1981.
[SmithJ83] Smith, James E., Andrew R. Pleszkun, Randy H. Katz, and James R. Goodman, "PIPE: A High Performance VLSI Architecture," Proceeding, IEEE International Workshop on Computer System Organization, pp 131-138, March 1983.

[Wegman84] Wegman, Mark N., Frank Kenneth Zadeck, "Constant Propagation with Conditional Branches," POPL 1984.

[Wulf75] Wulf, William, *et al.*, "The Design of an Optimizing Compiler," Elsevier Computer Science Library, 1975.

[Yershov71] Yershov, A.P., "The Alpha Automatic Programming System," Academic Press, London 1971.

[Young85] Young, Honesty C., "Evaluation of A Decoupled Computer Architecture and the Design of A Vector Extension," Computer Science Technical Report #603, July 1985.