Register Allocation and Code Scheduling for Load/Store Architectures

by

Wei-Chung Hsu

Computer Sciences Technical Report #722

October 1987

REGISTER ALLOCATION AND CODE SCHEDULING

FOR

LOAD/STORE ARCHITECTURES

by

WEI-CHUNG HSU

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN -- MADISON

1987

Abstract

To achieve high performance, the structure of on-chip memory in a single-chip computer must be appropriate, and it must be allocated effectively to minimize off-chip communication. Since the off-chip memory bandwidth of single-chip computers is severely limited, data caches that exploit spatial locality to achieve high hit rates are not appropriate. A register file, which can be managed by compilers, might be more effective than a data cache as an on-chip memory structure. With a load/store architecture, compilers can separate operand fetches from their use by scheduling code, thus achieving high hit rates without increasing memory traffic. Register allocation also exploits temporal locality better than a data cache does.

This thesis investigates how effective register allocation could be and studies the interdependency problem between register allocation and code scheduling. A model of perfect register allocation is explored. An algorithm for optimal local register allocation is then developed. Since the optimal algorithm needs exponential time in the worst case, a heuristic algorithm which has near-optimal performance is proposed and compared with other existing heuristic algorithms. Through trace simulation, the perfect register allocation model is shown to be much more effective in reducing off-chip memory traffic than cache memory of the same size.

Code scheduling interferes with register allocation, especially for large basic blocks. Two methods are proposed to solve this interference: (1) an integrated code scheduling technique; and (2) a DAG-driven register allocator. The integrated code scheduling method combines two scheduling techniques–one to reduce pipeline delays and the other to minimize register usage–into a single phase. By keeping track of the

number of available registers, the scheduler can choose the appropriate scheduling technique to schedule a better code sequence. The DAG-driven register allocator uses the Dependency DAG to assist in assigning registers; it introduces much less extra dependency than does an ordinary register allocator. Both approaches are shown to generate more efficient code sequences than conventional techniques in the simulations.

Acknowledgements

It has been my privilege and good fortune to work and study with Professor James Goodman for whom I have the greatest respect. I would like to take this opportunity to thank him for all his encouragement, support, and direction. His enthusiasm, encouragement and patience have made this thesis possible.

I would like to thank the other members of my committee: Charles Fischer, Marvin Solomon, Guri Sohi, and John Beetem. I learned how to write a compiler from Charles. Marvin taught me the optimizing compilers skills. Guri spent many hours discussing with me. John offered me extremely helpful comments.

Thanks are also due to my dear friends, especially, Phil Pfeiffer and Matt Farrens, for their help in revising my writing.

I would be amiss, if I didn't acknowledge how fortunate I've been to have a family that has given me the strength to follow my dreams. I would like to express my deepest gratitude to my wife, Diane, for her sacrifice and endurance over these many years.

Table of Contents

Chapter 7: Conclusions

List of Figures

List of Tables

Chapter 1
Introduction

1.1. On-Chip Memory

Advances in semiconductor technology have made it possible to design and fabricate an extremely high-performance CPU on a single chip. It will soon be possible to fabricate a single chip containing more transistors than the CPU of a modern supercomputer. However, the performance of these single-chip processors will be severely limited by the memory systems. This is because off-chip memory access is slow and, more importantly, the *pin bandwidth*— the rate at which data can be moved through the pins of the chip — is limited.

Many recent innovations in single-chip processors involve ways to bypass this rigid constraint on memory/processor interaction, primarily by introducing additional memory in one form or another on the processor chip. The on-chip memory can be accessed much faster than off-chip memory, and can supply a bandwidth as high as necessary. Unfortunately, the on-chip memory cannot be as large as needed for high-performance computers. In fact, the memory necessary to support a single-chip, high-performance processor will require many chips for the foreseeable future. So the on-chip memory can only be thought of as the top of the memory hierarchy.

A memory hierarchy is effective because of two kinds of locality of memory references: *spatial* and *temporal*. Spatial locality refers to the property that memory accesses over a short period of time tend to be clustered in space. Temporal locality refers to the property that references to a given location tend to be clustered in time. Using spatial locality is very effective in increasing the hit ratio of a memory hierarchy.

However, techniques for exploiting spatial locality, such as prefetching or using a large transfer unit often increase the bandwidth requirement of the memory system. Due to the limited pin-bandwidth, features of the on-chip memory that exploit spatial locality must be used carefully to avoid increasing the off-chip traffic. In other words, temporal locality should be favored in designing the on-chip memory.

1.2. Data Cache or Registers

Memory references can be divided into three kinds: instruction accesses, local variable accesses (i.e. the run-time stack), and global variable accesses. Since instructions exhibit temporal and spatial locality in a very consistent manner, some form of on-chip memory is clearly warranted, as suggested in several studies [SmitJ83, Moto82, Alpe83, Henn84]. The organization of such memory is also well-understood [SmitJ85].

In Chapter 2, a study of data reference behavior shows that local data accesses exhibit a very high temporal locality. This implies that local data, like instructions, is a good candidate for the on-chip memory. Global data accesses, on the other hand, rely primarily on spatial locality to achieve high hit rate. Due to the pin-bandwidth limitation, a large transfer unit may result in long latency times because the unit must be time-multiplexed over only a few pins. In addition, much of the transferred data may be useless (for example, when the stride of the array references is not one). The redundant traffic is harmful to the already limited bandwidth. One possible solution to this problem is to have a smart data cache which can correctly prefetch the needed data, thus achieving a high hit rate without increasing data traffic. However, it is hard to envision an efficient hardware prefetch algorithm which will handle intermixed

reference patterns. Even if there is such an algorithm, it may be costly to implement and may slow down the basic cache operations. Therefore, on-chip data caches for global data references are unlikely to be effective.

Unlike instruction memory, the data memory can be controlled to some extent by the programmer. For example, frequently used variables can be allocated to registers, reducing redundant address calculations and memory traffic. In addition, data can be preloaded into a register and referenced several instructions later. With a pipelined implementation, memory access operations can be overlapped with computation operations, effectively hiding the memory latency.

Ideally, this "exact prefetch" scheme which fetches exactly the data needed will always have a data "hit" without increasing off-chip data traffic. A load/store architecture (also referred to as register-oriented architecture; e.g. the Cray-1S [Cray82]) serves such a purpose. Such architectures encourages the heavy use of registers to decrease redundant memory traffic [Henn82], and are easy to pipeline [Kogg81]. Although programs for load/store architectures may be larger, which implies an increased instruction bandwidth, load/store architectures have significantly lower data memory bandwidth [Radi82, Henn84]. Lower data memory bandwidth is highly desirable since data access is less predictable than instruction access. Thus, it can be advantageous to trade increased instruction bandwidth for decreased data bandwidth. Load/store architectures have been adopted in some recent high-performance VLSI architectures [Henn84, Patt85, Radi82, Good85, Birn86, Neff86].

The use of a set of registers has additional advantages: (1) it outperforms a data cache by nearly a factor of two in both speed and cost [Ditz82]; (2) it potentially can use better replacement algorithms with information available at compile time to

allocate and purge data more effectively than a data cache; (3) registers make the coherency problem for multiple processors easier to deal with — registers should perhaps be thought of as local memory. The major problem with the use of a set of registers is that static (compile-time) binding may perform poorly when most of the information for management cannot be obtained until run-time. For example, memory aliasing and conditional branching often force compilers to perform conservative allocation and scheduling, decreasing the efficiency of the static management. However, improvements in compiler technology (i.e. anti-alias analysis [Nico85,Coop84,CoutB86] and software branch prediction [Fish81]) enable compilers to perform efficient allocation and scheduling for certain applications, notably scientific computations. For applications which make static binding more difficult, a data cache may be a better candidate, since the cache is better adapted to dynamic program behavior.

The other argument which favors data caches is that data caches are architecture-independent — they can be used to improve system performance without affecting programs. This is a more compelling argument for implementations of architectures defined before the availability of cache. In order to increase performance, many newer computers make the cache visible. For example, the IBM 801 [Radi81] and the HP Spectrum [Birn86] have 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 [Radi82]. Therefore, data caches are no longer transparent to programmers in some high-performance architectures. Although a visible cache can be more effective through the use of cache hints, we believe the effort necessary to provide effective hints to the cache is comparable to the effort necessary to allocate

registers effectively.

1.3. Research Problems

Load/store architectures simplify code generation in some aspects, but they need more optimizations to realize the full advantage of the architecture features [Birn86]. Two compiler optimization techniques, register allocation and code scheduling, are crucial to the effectiveness of a load/store architecture.

1.3.1. Register Allocation

Register allocation is a difficult but important function of an optimizing compiler. In the literature, the phrase "register allocation" has been used to describe two phases in a compilation: (1) register allocation, which refers to the decision of what names in a program should reside in registers; and (2) register assignment, which refers to the association of preallocated symbolic registers with real registers. Since the register file is the top of the memory hierarchy, it should be small and must be used efficiently. In this thesis, we are primarily dealing with the register assignment problem.

Local allocation refers to the assignment of registers over an entire block of straight-line code. An early exposition of this problem appears in the description of the Fortran I compiler [Back54, Cock70]. Horwitz *et al.* [Horw66] published a paper on this subject, giving an optimal algorithm for index register allocations. Kennedy [Kenn72] provided a refined version of Horwitz's algorithm. Freiburghouse [Frei74] proposed the "usage count" algorithm, compared it to some other heuristic algorithms, and concluded "usage count" is efficient for local allocation. His paper assumed the memory always keeps a correct copy of the contents in registers, therefore, Belady's

algorithm [Bela66] is optimal in his case. Fischer [Fisc87] gave an allocation algorithm that favors replacing clean registers to reduce the number of necessary store instructions. Although Horwitz's and Belady's algorithm are known to be optimal in local register assignment, they are inadequate for general purpose register allocations. Detailed explanations will be given in Chapter 3. A model for finding an optimal assignment, which generates a minimal number of loads and stores for spilling, has been established in Chapter 3. Rules for speeding up the computation have been provided.

Global allocation refers to the assignment of registers over a portion of the program that includes branches or high-level control constructs. A simple solution, known as "reference count" [Aho86], identifies frequently used data items (statically) and allocates them to registers over a portion of program (e.g. a loop), has been used in Fortran H compiler [Lowr69] and a Modula II compiler [Powe84]. More sophisticated approaches [Day70, Beat74, Kim78, Wulf75, Leve81] try to allocate more variables to registers by assigning a register to several variables which have disjoint lifetimes. Approaches [Chai81, Chow84] based on graph coloring model have claimed to be effective and elegant for global register allocation. However, instruction traces from IBM 801 and MIPS, which applied graph coloring allocation in their optimizing compilers, showed that there are still many load/store instructions compared to the trace from RISC [Patt85]. This is probably due to the increased cost of procedure calls: the more registers allocated, the more loads and stores needed for procedure calls to save and restore them. Wall [Wall86] proposed delaying register allocation until link time so that simultaneously live procedures will be allocated different sets of registers to avoid register saving and restoring. Two mutually exclusive procedures can share the

same set of registers. This is similar to a procedure-level graph coloring approach. Wall used 52 registers to perform the link time allocation. In Chapter 2, we study a model of "perfect" register allocation. Assuming all procedures are in-line expanded, all loops are fully unrolled, and all branches are correctly predicted, we would like to know how many registers can be effectively allocated. The effective number found is about 32, much smaller than the number in the RISC microprocessor, which is 138. The key point of effectively using a small number of registers is judicious spilling, a weak point of graph coloring allocation. We discuss the issue of register spilling in Chapter 2 and 3.

1.3.2. Code Scheduling

Code scheduling creates a semantically equivalent code sequence that has more overlapped operations and less wait time in a pipelined execution. It effectively hides latency of memory and functional units. Hiding latency has the effect of reducing pipeline interlocks. Code scheduling can be performed by hardware during execution, using dynamic code scheduling [Wiss84], or by the compiler at compile time, using static code scheduling. Dynamic code scheduling has been used in CDC 6600 and IBM 360/91. Due to its expense and its complexity, which may slow down the basic clock period, dynamic code scheduling has not been widely applied today. Research on static code scheduling includes [Arya85, Ausl82, Henn83, Youn85, Gibb86, Site78]. Arya formulates code scheduling as an integer programming problem [Arya85]. Auslander [Ausl82] describes code scheduling before register allocation (Prepass), while the others [Henn83, Gibb86, Site78] are concerned with scheduling after code generation and register allocation (Postpass). Prepass scheduling may introduce extra register

spilling since code scheduling lengthens pseudo-register lifetimes, increasing the number of simultaneously live registers. Postpass scheduling, on the other hand, may suffer from more restricted parallelism due to improper allocation of registers. This interdependence between register allocation and code scheduling has been studied in Chapter 5. Two solutions have been proposed and tested in Chapter 5 and 6.

1.3.3. Large Basic Blocks

Typically, basic blocks are small. For small basic blocks, performance differences between spilling algorithms are minuscule, and the interdependency problem is almost negligible. However, small basic blocks offer limited parallelism. Without sufficient parallelism, the latency of the memory or the function units cannot be hidden since there are not enough independent instructions to be executed while one instruction is waiting. With the increasing emphasis on parallel processing, especially on instruction level parallelism, researchers are looking for ways to obtain large basic blocks.

Usually, the smaller a unit (*e.g.*, a procedure, a basic block), the higher the overhead for unit transitions. For example, the procedure call overhead (saving and restoring registers) is proportionately higher for small procedures. Branches are relatively more expensive for small basic blocks in pipelined processors. Furthermore, small basic blocks often require more load/store instructions: Temporaries which reside in registers and are live on exit are stored to memory and reloaded when referenced in some later basic block. With large basic blocks, many such redundant loads and stores can be avoided.

Fifteen years ago, when saving memory space was the major concern, most optimizing compilers produced code that was as small as possible. Since small program

units permit more code sharing, they were favored. For example, the BLISS/11 optimizing compiler [Wulf75] tried to detect groups of similar expressions and replace them by a single subroutine. With decreasing memory cost and rapidly increased main memory size [Pohm83], static code size is not as important as before in determining system performance. In Chapter 4, techniques for generating large basic blocks are discussed. Those techniques that increase static code size but not dynamic code size are favored. In order to support pipeline/parallel processing, future optimizing techniques may favor large basic blocks, increasing the importance of this research.

Chapter 2
On the Use of Registers vs Data Cache to Minimize Memory Traffic

2.1. The Behavior of Local Accesses and Global Accesses

Recent research papers [e.g. Patt82, Hase85] have examined the use of on-chip memory to reduce local data accesses from off-chip memory. In order to understand how the on-chip memory could be used more effectively to reduce off-chip communications, we conducted an experiment to study the behavior of local data accesses and global data accesses.

To investigate data access locality, we used a trace-driven cache simulator. We generated and used five different traces for a VAX-11[1] architecture running UNIX[2] version 4.2BSD:

SORT The standard UNIX sorting program sorting the first 1000 entries in the on-line dictionary.

GREP The UNIX string matching program, searching through the dictionary for a string.

COMPACT
A program using an on-line algorithm which compresses files using an adaptive Huffman code.

CACHE A cache simulator program simulating a fully associative, write-through cache.

AS The standard UNIX (VAX-11) assembler translating an assembly program jmalloc.s.

---

[1] VAX is a trademark of Digital Equipment Company.

[2] UNIX is a trademark of AT&T.

The traces were collected on a VAX-11 by a program using the UNIX ptrace system call, which sets the VAX trace bit to trap after each instruction. The instruction is then interpreted and memory references are recorded. Thus the trace represents a single process executing without interruption. This is unrealistic in a time-sharing environment where frequent interruptions occur for task-switching and terminal handling. It is reasonable here because we are studying the locality characteristics of local and global accesses, not the effectiveness of a cache, which is profoundly affected by task switches. As we mentioned in the previous section, we are only interested in data caches. Therefore, the traces we used here include only data fetches and stores. In addition, the traces are split into two independent streams — global data accesses and stack accesses. Since some local variables may be allocated in registers rather than the stack by the standard C compiler (because of the use of register "hints" in C), we removed all register hint declarations from the source programs to force all local variables to be allocated in the stack.

Two cache parameters, total cache size and line size, were varied in this experiment. The other parameters we used were: fully associative placement algorithm, write-back policy, LRU replacement algorithm, four byte bus width, and write allocation policy (fetch-on-write [Smit82]) for a write miss. These parameters may be unrealistic, but are indicative of the best case. Traces are typically 100,000 instructions long so that cold start effects can be ignored.

2.1.1. Temporal Locality and Spatial Locality

Figure 2.1 shows that a small data cache for the stack can have a very high hit ratio by taking advantage only of temporal locality. This confirms observations of

others [Haik84]. A 128-byte cache with four-byte lines has a hit ratio as high as 98.6% on the average for stack accesses. Since the stack reuses the same storage, it has high and consistent temporal locality. An on-chip cache favors high and consistent temporal locality since a single-chip processor cannot have a wide bus due to the pin limitation.

In Figure 2.2, which is for global accesses, the miss ratio for larger line size is higher when the cache is small and is lower when the cache is large. When the cache size is fixed, a small line size means that there are more lines in the cache; that is, the cache can exploit more temporal locality. A large line size means that there are fewer lines in the cache, so that temporal locality is sacrificed for spatial locality. One interesting observation in this figure is that all the knee points occur when the cache has eight lines. Beyond the knee points, increasing the cache size has little effect on hit ratio. This implies that for global accesses, there are a few "hot spots" which contribute largely to the observed temporal locality. When the cache is small and the line is large, the number of lines in the cache is not large enough to catch all the "hot spots". Therefore the miss ratio is much higher than the cache with the same size but smaller lines. This is also known as "cache pollution" [Smit82].

According to Figure 2.2, it is possible to have a small data cache (smaller than 2K) with high hit ratio (over 95%) for global accesses. However, the cache must use large lines to exploit spatial locality to get the high hit ratio. From Figure 2.3, we learn that doubling the line size often doubles the bus traffic. Since the pin-bandwidth is limited, a large line size is undesirable for an on-chip cache.

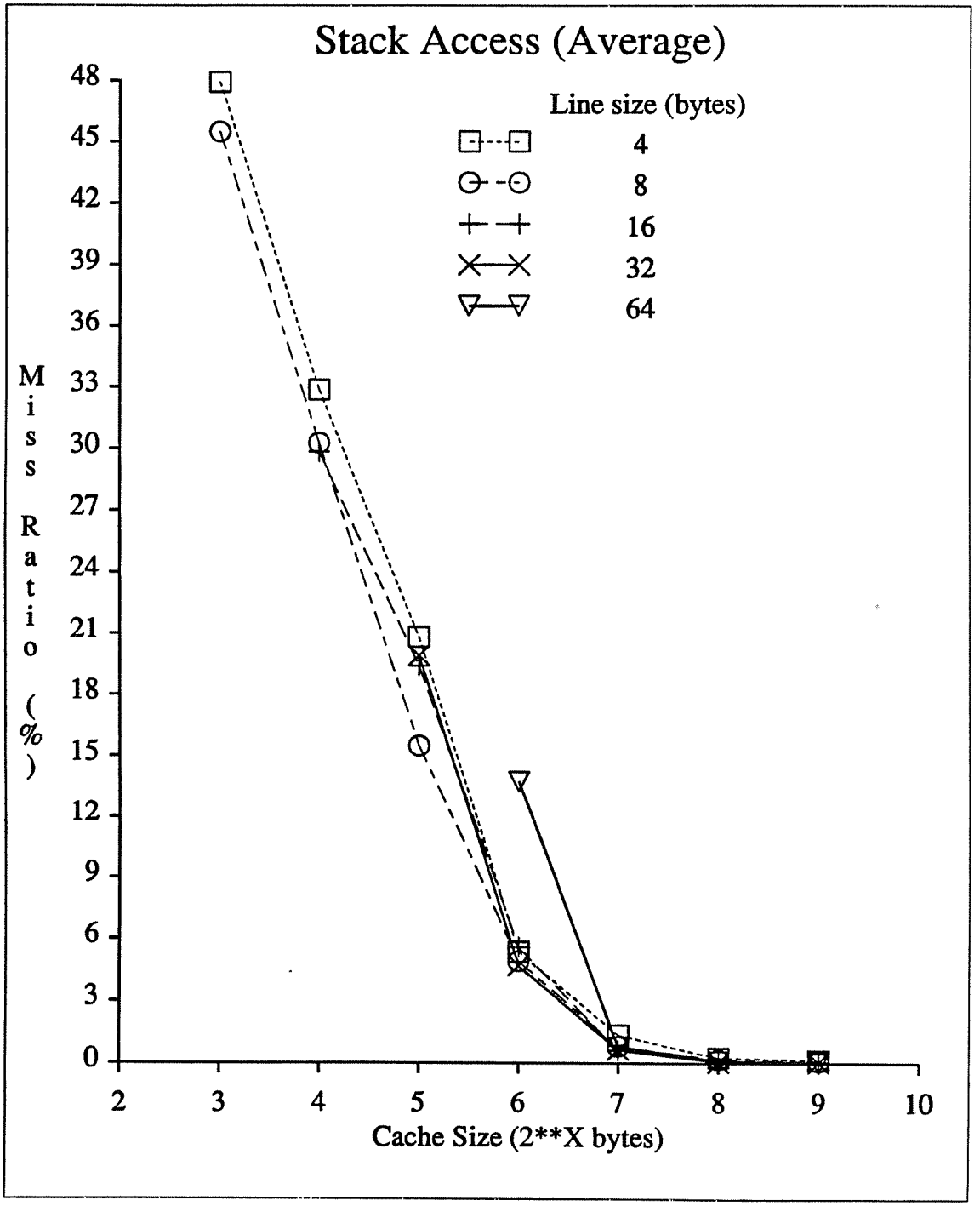


Figure 2.1 Cache Simulations for Stack Accesses

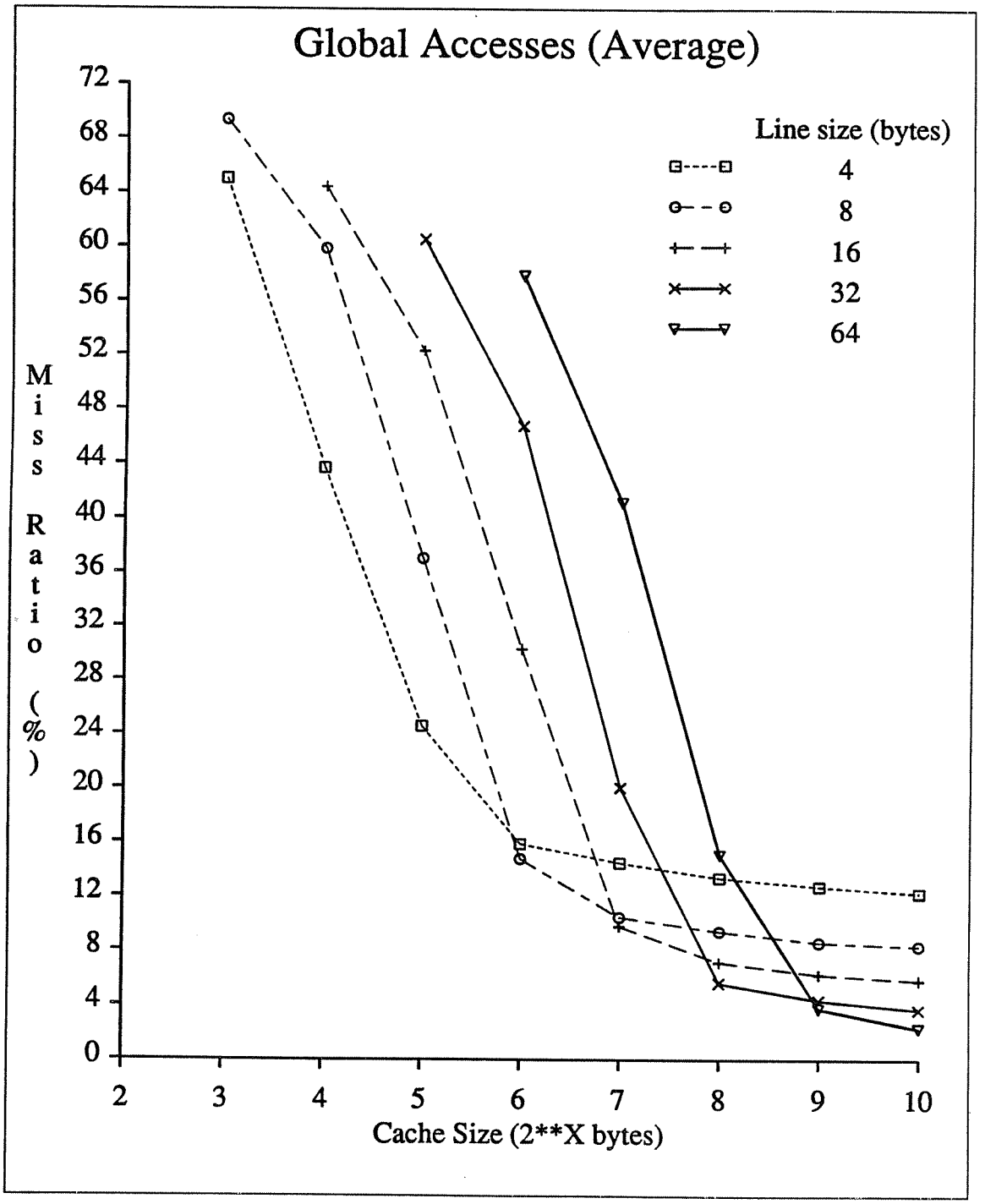


Figure 2.2 Cache Simulations for Global Accesses

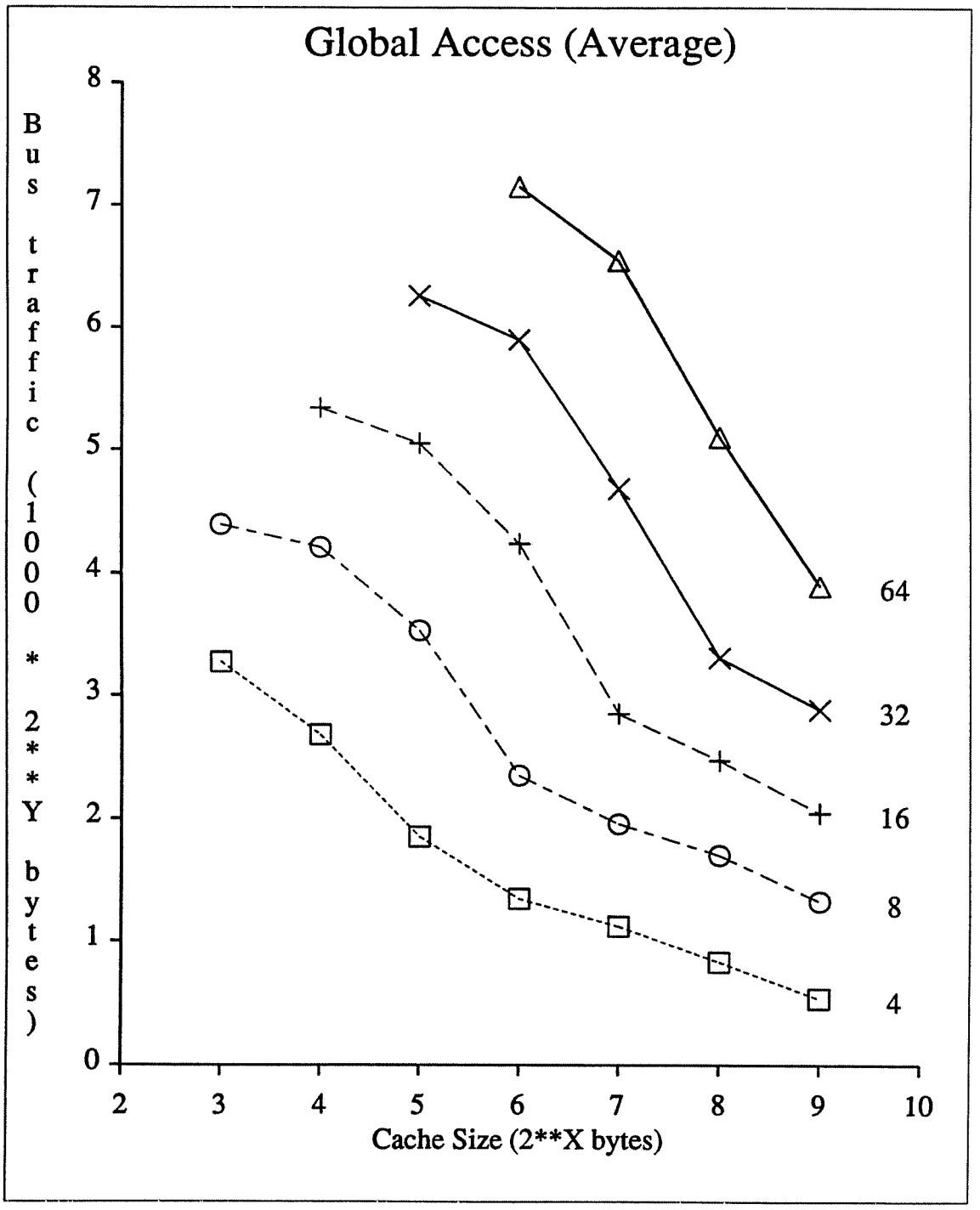


Figure 2.3 Cache Simulations for Global Accesses (Bus Traffic)

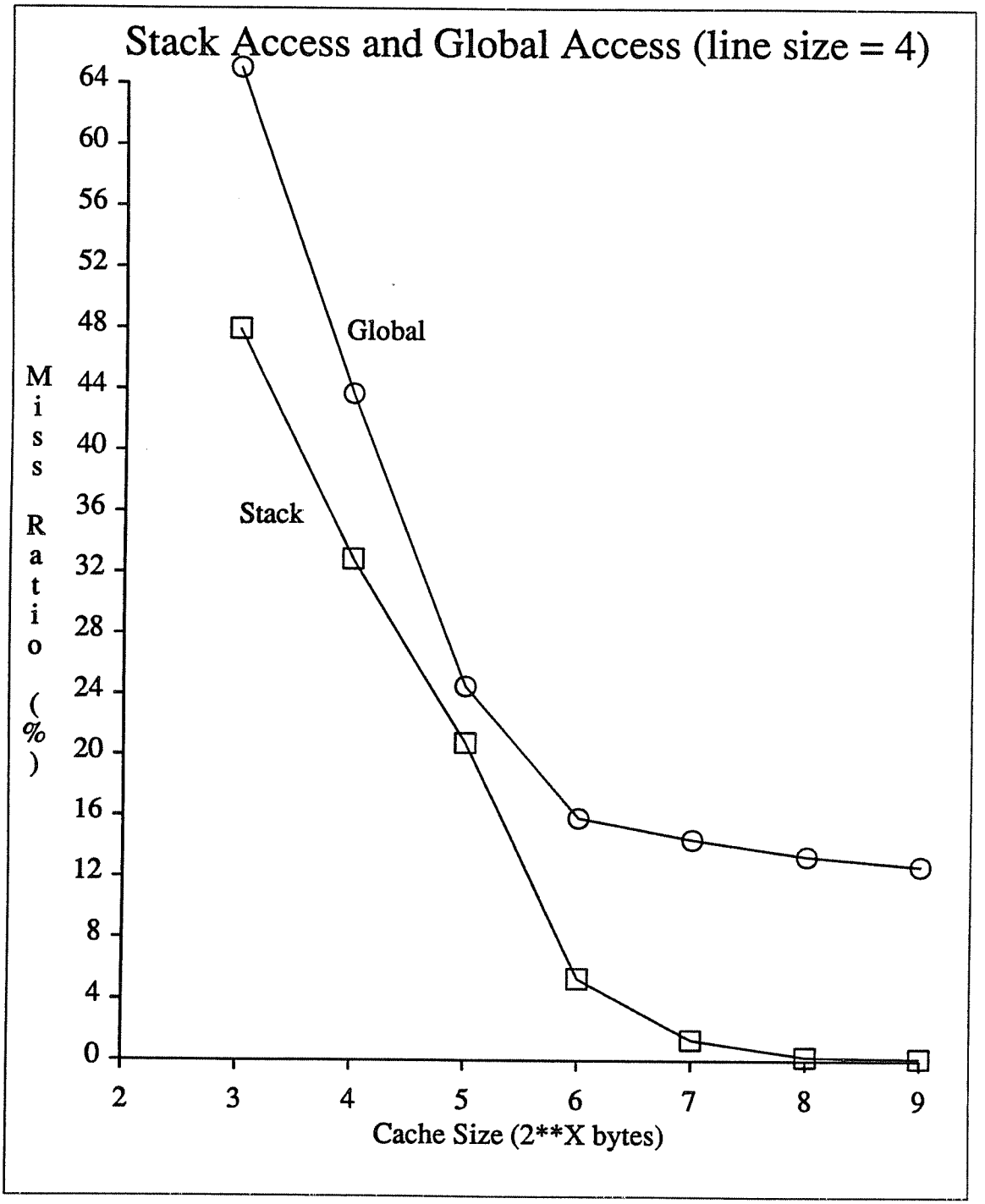


Figure 2.4 Comparison of Global Accesses to Stack Accesses

2.1.2. On-chip Memory for the Stack

We have found that an on-chip data cache for the stack can be very effective. Other on-chip memory structures are also possible for the stack. For example, the register stack and the Top-of-Stack (TOS) approaches. The register stack approach organizes an on-chip register file as a stack of register sets, allocated dynamically on a per procedure basis. This feature encourages compilers to allocate as many local variables as possible to the registers. Since each procedure call allocates a new register set, there are no memory accesses required to save/restore registers until the register stack overflows. It has been used by a wide range of machines (*e.g.* RISC I, C/70, Pyramid). The TOS approach automatically keeps the top of the stack in a high speed buffer. It has been exploited in the Symbolics 3600 machine. The C machine [Ditz82, Ditz87] demonstrates an elegant combination of the above two approaches. It works as a register stack with variable set size and preserves the semantics of the stack. Another interesting variation can be found in the Dragon computer [McCr84].

It is interesting to note the similarities between the "stack cache" and a data cache. The stack cache uses the LRU algorithm for replacement. Its (variable) line size is a stack frame. Only two address tags are required in the stack cache — the highest and the lowest address (MSP and SP) of data in the register stack. The stack cache has the advantage that replacements caused by pop up underflows (when a **catch** is executed) do not require the replaced frame to be flushed out to memory, since the frame is dead. Nevertheless, the stack cache is more restrictive than a conventional cache since it can only hold TOS variables. Non-TOS variables are likely to be active as well because procedures may pass their local variables as by-reference parameters to other procedures, though the compiler can allocate space for such variables on TOS if making

a redundant copy is allowed.

The stack cache (also a register stack) uses a relatively large line size — a frame. As we have discussed in the previous section, when a cache uses a large line size, it requires a larger size to capture the same number of active lines due to temporal locality. This explains why a register stack typically needs more entries. For example, equating four bytes of cache memory to one register, we found that 32 or 64 registers (*i.e.*, 128 or 256 bytes of cache memory) yielded 98.6% and 99.7% hit ratios respectively. Though it is possible to achieve such high hit ratios using 32 or 64 registers, RISC I uses 128 and the C/70 uses 1024 registers. Then why does the stack cache in CRISP have only 32 cache registers? This is because the CRISP optimizing compiler [Band87] performs stack compression to minimize the frame size (line size). We note that the top of the memory hierarchy should be very fast. The basic cycle time of the machine may well be constrained by register access, which is a function of the register file size [Henn84, Ditz86][3].

We feel that using small lines to exploit temporal locality is especially important for on-chip memory. Not only does it produce less bus traffic, but its fine granularity allows small storage to capture temporal locality more effectively (the samller the line, the more lines can be kept in the cache). For example, if only half of the variables in a procedure are active during a call (the other half may be used for error handling or in a path which is not executed during this call), it is wasteful to keep the whole frame on the cache. It may be argued that a high hit rate is also important to minimize the effect

[3] Ditzel reports that doubling the number of registers increases the basic clock cycle by roughly 30%.

of memory latency. However, if the small storage is a register file, then the latency can be overlapped with instruction execution in a pipelined processor by separating operand load from operand use, a common technique in supercomputers.

In order to use a small storage as effectively as a data cache for the stack without paying the cost of space overhead (*e.g.* address tags) and time overhead, we studied the use of a register file, managing it as a nearly perfect cache at compile time. Radin said [Radi82]:

> All the registers which the CPU can afford to build in hardware should be *directly* and *simultaneously* addressable. Stack machines, machines that hide some of the registers to improve Call performance, and multiple-interrupt-level machines all seem to make poorer use of the available registers.

There are similarities between the function of register allocation and the function of a cache. For example, in register allocation, the placement algorithm is essentially fully associative, the line size is a register, the replacement algorithm might be Belady's MIN [Bela66], the write policy is write-back, and so on. The compiler has the advantage of better algorithms, but the disadvantage that it lacks run-time information. If necessary, compilers can use profiling information collected from previous runs to optimize the code generation, as has been proposed in the Bulldog compiler [Elli85] as well as the Fortran I compiler [Back57]. In addition, using registers has the potential to reduce more memory traffic than using a data cache. Two examples will serve to illustrate this: (1) When a new activation record is pushed and the first local variable is initialized, a write miss might occur. If write allocation is used, a whole line will be fetched. Because this is a new activation record, all old values in the fetched line are useless, resulting in unnecessary fetches. Radin points this out as a motivation for cache hints. (2) When replacing a dirty line, a write operation is required. However, if

the dirty value will never be used again, then the write operation is not necessary. Caches cannot identify those "dead" lines while a compiler usually can. Note that even cache hints will not work in this case unless an entire block is dead. In register allocation, when a register is replaced, if the variable currently in the register is dead, no store instruction is needed.

One obvious question is how to deal with the register saving/restoring overhead across a procedure call. We will discuss this question in detail in the next section.

2.2. Register Allocation and Procedure Calls

An elegant method of register allocation has been developed by G. Chaitin, primarily for the PL.8 compiler in the IBM 801 project [Chai81, Ausl82, Radi82]. In this approach, the problem is formulated as a graph coloring problem: Each node in the graph stands for a computed quantity or a local variable that resides in a machine register, and two nodes are connected by an edge if the quantities 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 a similar model, supplemented with an enhanced spilling algorithm [Chow84]. Both graph coloring algorithms claim to work very well: rarely is there a 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 [*sic*] 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. [Patt85]

If the graph coloring algorithm requires little code spilling, why is the number of loads and stores so high? Apparently because of procedure calls.

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 to save/restore those registers when a procedure call or return is executed. The observation that register allocation increases the cost of procedure calls has been reported several times [*e.g.* Ditz82, Henn84]. Because of the register saving/restoring overhead, procedure calls become the most costly source language statement [Patt82]. Multiple register sets have been adopted in architectures such as RISC I, Pyramid [Raga83] to speed up procedure calls. Though the approach of multiple register sets is effective for eliminating the load/store overhead on procedure calls, this technique introduces new problems: the large number of registers may slow down the basic cycle time, consume a large area of silicon, and increase process switching time.

Again quoting Patterson,

> 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 [Patt85].

We are therefore interested in knowing if compiler technology can achieve a more effective use of the chip area than that of a more complex architectural scheme.

Modern compilers use procedure integration, or *in-line expansion* to reduce the procedure call overhead [Alle80, Ausl82, Chow83, Macl84]. Procedure integration is a

good program optimization technique though it may increase program size. It can reduce procedure calls, create larger basic blocks, remove parameter passing overhead, remove dead code and perform some computation at compile time through constant propagation, and allow better global optimizations [Alle72, Alle80, Sche77, Macl84, Ball79]. The experimentation with complex instructions in the 70s was an attempt to introduce the notion of an assembly language-level procedure corresponding to a high-level language statement. The recent interest in simple instructions [Radi82, Henn84, Patt85] is in part motivated by the observation that in-line expansion of these procedures (*i.e.*, compiling a program down to the micro-instruction level) reduces the number of steps in the execution of the program. Although in-line expansion increases the code space if procedures are called more than once (statically), it usually increases only the static code size, not the dynamic memory bandwidth (the latter is especially important for designing VLSI processors with instruction caches). In addition, modern programming practice encourages the use of many small procedures, many of which may be called from exactly one place. 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 [Sche77]. When an on-chip instruction cache is used, in-line expansion must be done more carefully, since a procedure that is called more than once in a loop may overflow the cache after expansion. This suggests a judicious in-line expander which takes the size of the instruction buffer into account.

Procedure integration can reduce the number of procedure calls, but it cannot remove register saving/restoring overhead entirely. In-line expansion creates many more local variables in the calling procedure, and increases 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 saving/restoring overhead, the procedure integration technique may merely shift the problem from procedure calls to spilling. Alternatively, it allows a compiler to make effective use of more registers, *if they are available* [Radin82].

One weak point of the graph coloring algorithm is that it does not handle spilling very well. The spilling process (at compile time) is slow and may generate many more loads/stores than necessary. When the coloring algorithm is blocked, it will spill the node for which the cost of spilling is the smallest (where the cost is defined as the estimated number of references to 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 loads/stores [Hsu85].

Chaitin claims spilling is not a problem, because (1) it occurs rarely; and (2) it converges quite rapidly [Chai82]. However, if we want to reduce the overhead of procedure calls by procedure integration, then 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. Therefore, allocation algorithms based on graph coloring will not work well with procedure integration techniques. In other words, procedure integration works by increasing use of registers until spilling occurs.

If we have a very large number of registers available, say 1024, then the performance of a spilling algorithm is perhaps not so important. However, VLSI

constraints force the most effective use of limited area. Therefore, we need a model which is amendable to efficient spilling.

2.3. A Model for Perfect Register Allocation

We have explained that using a set of registers may reduce bus traffic better than a data cache for the stack. In this section, a model is introduced to examine the performance of an optimal register allocation compared to a near-perfect data cache for the stack, and to study the degree of performance degradation resulting from conventional compilers. If the performance difference is insignificant, further research in this area may not be worthwhile. A load/store architecture is used and a "perfect" register allocation is defined as follows:

(1) Every memory word can be allocated to a register.

(2) Future memory reference information is known.

(3) Registers can be dynamically allocated (e.g. a variable can be allocated in different registers at different iterations of a loop).

2.3.1. How Realistic is the Perfect Register Allocation?

There is a gap between the perfect allocation model and today's practice in compilers. With improving compiler techniques and innovative architectural features, the gap could be reduced. We discuss the gap as follows.

(1) Non-scalar variables are not easily allocated to registers. For example, arrays and strings are typically assigned to primary memory. However, with appropriate architecture support, non-scalar variables can be allocated in registers. For example, arrays can be allocated in vector registers [Cray82]. Moreover, a section

of memory can be allocated in contiguous registers, for example, Cray-1 has instructions to move blocks of data from memory to B or T registers.

Possible memory aliases—a data item can be reached through different names (or pointers)—often prevent variables from being allocated in registers. With improving anti-alias analysis techniques [Nico85, Coop84, CoutB86], this restriction will become less severe. In-line expansions can also reduce much of the alias problem caused by parameter passing.

(2) Due to conditional branches and loop structures, only a limited amount of information concerning future references can be obtained at compile time. Nevertheless, through the use of loop unrolling, code replication, and branch prediction, the available information concerning future references can be increased.

(3) Usually, variables are bound to registers statically. Once the instruction has been generated, the register designation cannot be changed at run time. For example, in a loop, if variable A is allocated in register 1, the instruction to reference A is bound to register 1 and can not be rebound to other registers from one iteration to another. (Notice that the model essentially assumes all loops are fully unrolled and all procedures are in-line expanded) If we unroll the loop several times, it is possible to bind the different instances of variable A with different registers.

2.3.2. An Initial Evaluation

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 should be an extension to Horwitz's algorithm, which optimizes the bus traffic.

Although in the literature, Horwitz's algorithm [Horw66] was referenced as the algorithm for minimum bus traffic, it is inadequate in our case. This will be discussed in detail in the next chapter. Two important performance metrics are often used in cache simulations: *miss ratio* and *bus traffic*. It is more precise to use the *bus traffic* as the performance measurement in our case. However, measuring the optimal bus traffic is much harder than measuring the optimal miss ratio (as will be shown in the next section). On the other hand, the Belady's MIN algorithm, which optimizes the miss rate, is simple, and fast. Therefore, we choose it to do the initial evaluation. This is not too unreasonable, since a lower miss ratio often means less bus traffic. In the next section, after the introduction of the algorithm for optimal bus traffic and its heuristic version, we will use them to reevaluate the perfect allocation model.

The traces are the same as in the previous section. For this experiment, "perfect" register allocation can achieve only the minimal miss ratio (since MIN is used), not the minimal bus traffic. In the initial study, "perfect" register allocation results in 15% ~ 35% less bus traffic than a data cache with the same size using LRU replacement algorithm. This initial evaluation is encouraging. In the next section, we will show that much more bus traffic can be eliminated by using the algorithm that optimizes the bus

traffic.

The performance of the "perfect" register allocation is also compared to conventional compilers, the UNIX C compiler and the Modula-2 compiler [Powe84]. From the 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 are available for allocation in VAX-11, but 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 *sort* program, 53% for *grep* and 32% for *cache*. The Modula-2 compiler performs more optimizations than 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).

Due to inaccessibility, we have not compared to some state of the art optimizing compilers, for example, the PL.8 compiler [Ausl82], the UOPT [Chow83], and the Wall's approach for interprocedure allocation [Wall86]. Wall claims that 60% to 90% of the loads and stores of scalar variables can be removed using his link-time allocation scheme with 52 registers.

The perfect register allocation shows a great potential of using registers to effectively decrease bus traffic. This potential has not been fully utilized by the conventional compilers. As will see in the next chapter, it can perform even better by saving unnecessary stores.

Chapter 3
Optimal Allocation in Basic Blocks

In 1966, Horwitz *et al.* [Horw66] published a definitive paper on index register allocation in straight-line programs. Their algorithm minimizes the number of loads and stores. Later algorithms [Lucc67, Kenn72] are mainly improvements of Horwitz's. Horwitz reduces the problem of index register allocation to that of finding a "shortest path" through a WDAG (Weighted Directed Acyclic Graph). In order to prevent the WDAG from growing too rapidly, Horwitz provides rules to restrict the growth. The weights that Horwitz uses in the WDAG are based on two basic operations 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 "ADD r_3, r_1, r_2" reads registers r_1 and r_2 and writes register r_3), the rules in Horwitz's algorithm are inadequate for general purpose register allocation.

3.1. The Model

We modify the cost function of Horwitz's model and build a new set of rules suitable for general purpose registers. Our model is based on a load/store machine architecture.

$X = \{r_1, r_2, ..., r_M\}$ is a set of pseudo-registers. We assume that, at the allocation phase, pseudo-registers are assigned to temporaries, local variables, frequently used constants, etc. Later in the assignment phase, a register allocator maps pseudo-registers to real registers. This technique has been used widely [Leve81, Kim78, Ausl82, Madh82,

Davi84, CoutA86]. A pseudo-register is an *allocated* pseudo-register if it is currently associated with one real register, otherwise it is *unallocated.* We assume that each pseudo-register has an associated memory location in which to store its content.

S = {clean, dirty} is a set of two states. The clean state means that the value of an allocated pseudo-register is consistent with the value in the pseudo-register's corresponding memory location. The dirty state means the value of an allocated pseudo-register is not consistent with its value in memory.

$Q_i = (q_i^1, q_i^2 ..., q_i^N)$ is a register configuration, where N is the number of real registers and q_i^j belongs to X × S. We assume that in the initial configuration Q_0, each q_0^j is a pseudo-register that will never be used.

P = $\{I_1, I_2, ..., I_n\}$ is a sequential program, a sequence of pseudo-register operations. There are two operations: read and write. A pseudo-register must be allocated before the operations are applied to it. A read from an allocated pseudo-register will not change its state, while a write to it will put it in a dirty state.[1] A program here means a sequence of pseudo-register reads and writes.[2] The *i*th symbol of the program is called

---

[1] A load instruction, which copies data from the memory to a register, by definition, is a write operation to registers. Thus, a load instruction will put the allocated pseudo-register in a dirty state. However, if the load instruction copies data from the pseudo-register's corresponding memory location to the pseudo-register, then the pseudo-register should be in a clean state. Therefore, a load from the pseudo-register's corresponding memory location to the pseudo-register is treated as a read rather than a write.

[2] 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 is useful for making good replacement decisions. Simple extensions to accommodate such complications have been made in one of our implementations.

the *i*th step. As in Horwitz's paper [Horw66], we use r to denote a read from pseudo-register r, and r* to denote a write to r. For the register configuration, we also use r to denote that pseudo-register r is in a clean state and r^* to denote that r is in a dirty state. We can specify an example program as follows :

$$r_2^* \; r_1 \; r_2 \; r_3^* \; r_0 \; r_3$$

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

```
MOVE    r2,   #1
ADD     r3,   r1,   r2
STORE   r3,   X(r0)
```

$A = (Q_1,..,Q_n)$ is a register allocation for a program P (of n steps). It is a sequence of register configurations.

If the pseudo-register accessed at the *i*th step is in the (*i-1*)st configuration (*i.e.* the pseudo-register is allocated), it is said to be a *hit*. If not, it is said to be a *miss*. No loads or stores are required for a hit. A read miss requires one load to fetch data from the memory location into a real register. A write miss needs no loads. Register replacement is required for a miss. When replacing a dirty pseudo-register, one store is required to update its corresponding memory location. However, if the dirty pseudo-register is dead (*i.e.* its content will never be used again), update is not necessary.

We define two distance functions: (1) NEXTRD(i,x) returns the number of steps from step i to the first step after i that reads pseudo-register x; and (2) NEXTWR(i,x) returns the number of steps from step i to the first step after i that writes x. If no instance of the appropriate access is found, the functions return ∞. An allocated pseudo-register x is said to be DEAD at step i if: (1) NEXTRD(i-1,x)=∞ (x will never

be read again) or (2) NEXTWR(i-1,x) < NEXTRD(i-1,x) (the next access of x is a write). Q_{i-1} and Q_i are either equivalent or different in only one component. 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 ≠ x′. We define the cost of a configuration change, Cost(Q_{i-1}, Q_i), as follows:

A) Store cost:

```
If s=clean
      then    cost = 0
else If s=dirty and x is DEAD
        then cost = 0
   else cost = 1
```

If a register is clean, no register store is needed as its value is already in the corresponding memory location. If the register is dirty and dead, there is no need to store the value; if the register is dirty and live, one store instruction is needed to update memory.

B) Load cost:

```
If s′=clean
      then      cost = 1
else If s′=dirty
        then      cost = 0
```

The state s′ is clean means this is a read miss. A read miss needs one load instruction to fetch data from memory into a register.

The state s′ is dirty means this is a write miss. A write miss needs no fetch.

Cost(Q_{i-1}, Q_i) is the sum of the Store cost and the Load 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 Horwitz's model, because the basic operation for writing an index register is a modification, it does.

(2) In our model, if the next access of a pseudo-register x is a write, the replacement of x needs no stores to update the memory. This never happens in Horwitz's model because there are no direct write operations.

With our model, index register allocation [Horw66, Kenn72] is a special case of general purpose register allocation in which a write to pseudo-register r is always preceded by a read of pseudo-register r.

The cost of a register configuration Q_j is defined as:

$$\text{Cost}(Q_j) = \sum_{i=1}^{j} \text{Cost}(Q_{i-1}, Q_i)$$

The cost of an allocation A of a program with n steps is defined as:

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

A WDAG can be constructed as follows: For each step of the program we associate a node of the graph with each configuration that may occur at this step; the given initial configuration is the only one associated with step zero. We can then draw a branch from any node associated with the ith step to any node associated with the $(i+1)$-st step, giving this branch a weight which is the cost of getting from the first configuration to the second. The problem of minimizing Cost(A) can then be considered as finding the shortest path in this graph from the initial node to some node

associated with the last step of the program. That is, we find an register allocation *A* which generates minimal number of loads and stores.

Example

Assume there are only two real registers.

The input program is: $r_1^* \, r_2 \, r_3 \, r_2^* \, r_1$

The complete WDAG is shown in Figure 3.1 (costs of allocation are shown in parentheses, costs of configuration changes are associated with edges). The optimal allocation of this example is obtained from the rightmost path with a cost of 2.

3.2. Finding a Shortest Path in the WDAG

The search of a shortest path in the WDAG can be conducted by the construction of a search path tree [Sedg83]. The obvious approach to finding such a shortest path (an optimal allocation) is to enumerate all possible (legal) allocations for a given program, and to pick the one with least cost. Since the number of different configurations is finite for fixed M (M is the number of pseudo-registers) and N (N is the number of real registers), the search tree will not grow exponentially propotional to n (n is the number of steps). It will be bounded by the maximal number of different configurations. However, the number of different configurations grows exponentially with N and M. In a legal allocation, the pseudo-register at the *i*th step must be in the *i*th configuration to become an allocated pseudo-register. And each allocated pseudo-register can be in a clean or a dirty state. The number of different configurations, therefore, is C(M-1, N-1)$\times 2^N$. From Table 3.1, which is generated from this formula,

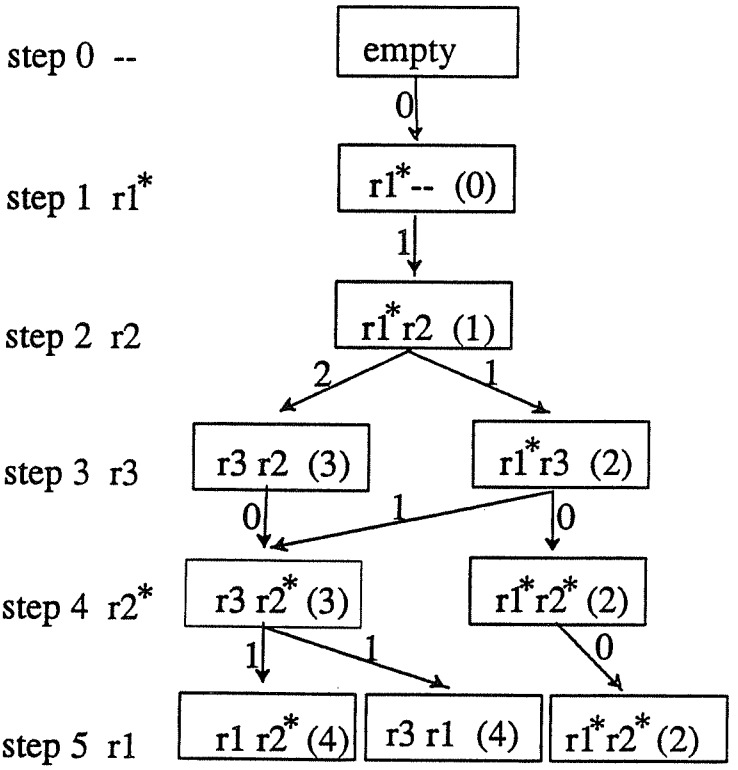


Figure 3.1 WDAG for Optimal Register Allocation

we see that the search tree method is computationally infeasible even for moderate values of M and N. Horwitz *et al* [Horw66] provided some rules to restrict this exponential growth; we shall do likewise.

3.3. The Rules for Pruning the WDAG

In order to make this model computationally feasible, we define a set of rules to prune the WDAG. A configuration Q_i is partitioned into four disjoint sets as follows:

set 1 (DEAD set) =
{ x or x^* | x in Q_i, x is DEAD }

set 2 (CLEAN set) =
{ x | x in Q_i, NEXTRD(i,x) < NEXTWR(i,x) } This set consists of all the variables which are clean and will be read after step i before being written.

(M) Number of Pseudo-registers	(N) Number of Real 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 3.1 Rapid Growth of Register Configurations

set 3 (DIRTY-DEAD set) =
{ x^* | x in Q_i, NEXTRD(i,x) < NEXTWR(i,x) and x is DEAD after step i+NEXTRD(i,x) } This is a set of pairs that exhibit the reference pattern: $x^* \cdots x \cdots x^*$ or $x^* \cdots x \cdots \infty$.

set 4 (DIRTY-LIVE set) =
{ x^* | x in Q_i, NEXTRD(i,x) < NEXTWR(i,x) and x is not DEAD after step i+NEXTRD(i,x) } This is a set of pairs that exhibit the reference pattern $x^* \cdots x \cdots x$.

Observations:

(1) Replacing an element from the DEAD set is always cheaper than replacing elements from other sets. This is obvious since the cost of replacing a dead register is 0.

(2) Among elements in the CLEAN set, replacement of the most distant one gives minimal cost. This is the Belady's MIN algorithm.

Proof:

Suppose allocated pseudo-registers x and y in Q_i, operations $I_j = x, I_k = y$, such that i < j < k, and that there is no $x \in \{ I_n \mid n=i+1, j-1 \}$, and no $y \in \{ I_n \mid n=i+1,k-1\}$. Assume I_{i+1} is not in Q_i, and one of registers x, y will be replaced. Two possible configurations, Q_{i+1}^1 and Q_{i+1}^2, are produced at step i+1, and they differ in exactly one element, $x \in Q_{i+1}^1$ and $y \in Q_{i+1}^2$. Let us isolate the remaining N-1 elements from x and y so that they are replaced under the same replacement strategy. For example, if an element z is replaced from Q_{i+1}^1 then the z in Q_{i+1}^2 must be replaced too; if x is replaced from Q_{i+1}^1 then y must be replaced from Q_{i+1}^2. At step j, there are two configurations: Q_{j-1}^1, which comes from Q_{i+1}^1, and Q_{j-1}^2 which comes from Q_{i+1}^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+1 to step j-1.

At step j, there are two cases:

(A) x is in Q_{j-1}^1 : this is a hit for Q_{j-1}^1.

Since there are no cost for a hit,

$$\text{Cost}(Q_j^1) = \text{Cost}(Q_{j-1}^1) \qquad (1)$$

Because Q_{j-1}^2 is originated from Q_{i+1}^2 and there is no $x \in \{I_n \mid n=i+1, j-1\}$, x will not be in Q_{j-1}^2. This is a miss for Q_{j-1}^2. One load instruction is required to fetch x and a store cost of U (0 or 1, depending whether the replaced element, say z, is clean or dirty) for replacement.

$\text{Cost}(Q_j^2) = \text{Cost}(Q_{j-1}^2) + 1 + \text{U}$ (2)

(I) If the replaced element $z = y$:

Then U = 0 and Q_j^2 has the same configuration as Q_j^1.

Because

$$\begin{aligned} \text{Cost}(Q_j^2) &= \text{Cost}(Q_{j-1}^2) + 1 \\ &= \text{Cost}(Q_{j-1}^1) + 1 \\ &= \text{Cost}(Q_j^1) + 1 \end{aligned}$$

Therefore, $\text{Cost}(Q_j^1) < \text{Cost}(Q_j^2)$. Hence, replacing y at step i is cheaper.

(II) If the replaced element $z \neq y$

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.

Because the cost of changing configuration Q_j^1 to Q_j^2 is (1 + U), where 1 is the cost to fetch y and U is the cost to replace z, and

$$\begin{aligned} & \text{Cost}(Q_j^1) + (1 + \text{U}) \\ = \; & \text{Cost}(Q_{j-1}^1) + 1 + \text{U} && \text{from (1)} \\ = \; & \text{Cost}(Q_{j-1}^2) + 1 + \text{U} \\ = \; & \text{Cost}(Q_j^2) && \text{from (2)} \end{aligned}$$

Therefore, if there exists an optimal cost path through Q_j^2, then there will be an optimal path through Q_j^1. Hence, replacing y at step i is not worse than replacing x.

(B) x is not in Q_{j-1}^1.

If x was replaced during the configuration changes from Q_{i+1}^1 to Q_{j-1}^1,

then y must have been replaced during the changes from Q_{i+1}^2 to Q_{j-1}^2. Therefore, Q_{j-1}^1 and Q_{j-1}^2 have the same configuration. Both Q_{j-1}^1 and Q_{j-1}^2 have a miss at step j. By replacing the same element, Q_j^1 and Q_j^2 must have the same cost. Hence, replacing y at step i is not worse than replacing x.

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

(3) If there exists a dirty register x^* and a clean register y in Q_i such that NEXTRD(i,x) < NEXTRD(i,y) (the clean element y has distance greater than the dirty element x), then replacing y is always cheaper.

Proof:

Suppose allocated pseudo-registers x^*, y in Q_i, operations $I_j = x$, $I_k = y$, such that i < j < k, and that there is no $x \in \{ I_n \mid n=i+1, j-1 \}$, and no $y \in \{ I_n \mid n=i+1,k-1\}$. Assume I_{i+1} is not in Q_i, and one of registers x^*, y will be replaced. Two possible configurations, Q_{i+1}^1 and Q_{i+1}^2, are produced at step i+1, and they differ in exactly one element, $x^* \in Q_{i+1}^1$ and $y \in Q_{i+1}^2$. Since one store instruction is required to update x's memory location,

$\text{Cost}(Q_{i+1}^2) = \text{Cost}(Q_{i+1}^1) + 1$.

Once again, we isolate the remaining N-1 elements from x^* and y so that they are replaced under the same replacement strategy. As described before, Q_{j-1}^1 comes from Q_{i+1}^1, and Q_{j-1}^2 comes from Q_{i+1}^2.

At step j, there are two cases:

(A) x^* is in Q_{j-1}^1 : this is a hit for Q_{j-1}^1.

x^* is in Q_{j-1}^1 implies y is in Q_{j-1}^2.

Since $\text{Cost}(Q_{i+1}^2) = \text{Cost}(Q_{i+1}^1) + 1$ and during the configuration changes, the identical elements from both configurations are replaced. Therefore,

$$\text{Cost}(Q_{j-1}^2) = \text{Cost}(Q_{j-1}^1) + 1 \qquad (3)$$

Because there are no cost associated with a hit,

$$\text{Cost}(Q_j^1) = \text{Cost}(Q_{j-1}^1) \qquad (4)$$

Because there is a miss for Q_{j-1}^2, one load instruction is required to fetch x and a store cost of U (0 or 1) to replace an element z.

$$\text{Cost}(Q_j^2) = \text{Cost}(Q_{j-1}^2) + 1 + \text{U} \qquad (5)$$

(I) If the replaced element $z = y$:

Then U = 0 and Q_j^2 has the same configuration as Q_j^1 except that x is dirty in Q_j^1 but clean in Q_j^2. It takes one store instruction to change a dirty state to a clean state.

$$\begin{aligned} \text{Cost}(Q_j^2) &= \text{Cost}(Q_{j-1}^2) + 1 \\ &= (\text{Cost}(Q_{j-1}^1) + 1) + 1 && \text{from (3)} \\ &= \text{Cost}(Q_j^1) + 2 && \text{from (4)} \end{aligned}$$

To make Q_j^1 the same configuration as Q_j^2, one store instruction is required to change x from dirty state to clean state. Because $\text{Cost}(Q_j^1)+1 < \text{Cost}(Q_j^2)$, replacing y at step i is cheaper.

(II) If the replaced element $z \neq y$

To make Q_j^1 the same configuration as Q_j^2, z in Q_j^1 should be

replaced by y and x in Q_j^1 should be changed from dirty to clean. This costs (2+U), one for fetching y, one for cleaning x, and U for replacing z.

$$\begin{aligned} & \text{Cost}(Q_j^1) + (2 + \text{U}) \\ = \quad & \text{Cost}(Q_{j-1}^1) + 1 + (1 + \text{U}) \qquad \text{from (4)} \\ = \quad & \text{Cost}(Q_{j-1}^2) + 1 + \text{U} \qquad \text{from (3)} \\ = \quad & \text{Cost}(Q_j^2) \qquad \text{from (5)} \end{aligned}$$

Therefore, if there exists an optimal cost path through Q_j^2, then there will be an optimal path through Q_j^1. Hence, replacing y at step i is not worse than replacing x.

(B) x is not in Q_{j-1}^1.

If x^* was replaced during the configuration changes from Q_{i+1}^1 to Q_{j-1}^1, then y must be replaced during the changes from Q_{i+1}^2 to Q_{j-1}^2. Therefore, the cost of configuration changes from Q_{i+1}^1 to Q_{j-1}^1 is one higher than the cost of configuration changes from Q_{i+1}^2 to Q_{j-1}^2.

$$\begin{aligned} \text{Cost}(Q_{j-1}^2) \quad = \quad & \text{Cost}(Q_{i+1}^2) + \sum_{k=i+2}^{j-1} \text{Cost}(Q_{k-1}^2, Q_k^2) \\ = \quad & \text{Cost}(Q_{i+1}^1) + 1 + (\sum_{k=i+2}^{j-1} \text{C}(Q_{k-1}^1, Q_k^1)) - 1 \\ = \quad & \text{Cost}(Q_{i+1}^1) + \sum_{k=i+2}^{j-1} \text{C}(Q_{k-1}^1, Q_k^1) \\ = \quad & \text{Cost}(Q_{j-1}^1) \end{aligned}$$

Now, Q_{j-1}^1 and Q_{j-1}^2 have the same configuration and with same cost. Both Q_{j-1}^1 and Q_{j-1}^2 have a miss. Replacing the same element, both Q_j^1 and Q_j^2 have the same cost. Hence, replacing y at step i is not worse

than replacing x.

From the above, it is clearly better to displace the clean variable with a greater distance.

(4) If x ∈ DIRTY-DEAD set (set 3) and y ∈ DIRTY sets (both set 3 and 4) such that NEXTRD(i,x) < NEXTRD(i,y), then replacing y is always cheaper.

Proof:

Suppose allocated pseudo-registers x^*, y^* in Q_i, operations $I_j = x, I_k = y$, such that i < j < k, and that there is no $x \in \{ I_n \mid n=i+1, j-1 \}$, and no $y \in \{ I_n \mid n=i+1,k-1 \}$. x is DEAD after step j. Assume I_{i+1} is not in Q_i, and one of dirty registers x^*, y^* will be replaced. Two possible configurations, Q_{i+1}^1 and Q_{i+1}^2, are produced at step i+1, and they differ in that $x^* \in Q_{i+1}^1$ and $y^* \in Q_{i+1}^2$. Because both x^* and y^* are dirty, $\text{Cost}(Q_{i+1}^2) = \text{Cost}(Q_{i+1}^1)$. The remaining N-1 elements are separated from x^* or y^* so that they should be replaced under the same replacement strategy. As described before, Q_{j-1}^1 comes from Q_{i+1}^1, and Q_{j-1}^2 comes from Q_{i+1}^2.

At step j, there are two cases:

(A) x^* is in Q_{j-1}^1: this is a hit for Q_{j-1}^1.

Since $\text{Cost}(Q_{i+1}^2) = \text{Cost}(Q_{i+1}^1)$ and the same elements are replaced during the configuration changes from Q_{i+1} to Q_{j-1}, therefore,

$$\text{Cost}(Q_{j-1}^2) = \text{Cost}(Q_{j-1}^1) \qquad (6)$$

Because there is a hit for Q_{j-1}^1,

$$\text{Cost}(Q_j^1) = \text{Cost}(Q_{j-1}^1) \qquad (7)$$

Because there is a miss for Q_{j-1}^2, one load instruction is required to fetch x and a store cost of U (0 or 1) to replace an element z.

$$\text{Cost}(Q_j^2) = \text{Cost}(Q_{j-1}^2) + 1 + U \qquad (8)$$

(I) If the replaced element $z = y^*$:

Then U = 1. Q_j^2 has the same configuration as Q_j^1 except that x is dirty in Q_j^1 but clean in Q_j^2. Since x is DEAD after step j, there is no difference in terms of store cost between replacing a dirty x and a clean x after step j. Therefore, Q_j^1 and Q_j^2 can be treated as the same configuration. In other words, it takes no additional cost to make Q_j^1 and Q_j^2 equivalent.

$$\begin{aligned} \text{Cost}(Q_j^2) &= \text{Cost}(Q_{j-1}^2)+1+1 && \text{from (8)} \\ &= \text{Cost}(Q_{j-1}^1) + 2 && \text{from (6)} \\ &= \text{Cost}(Q_j^1) + 2 && \text{from (7)} \end{aligned}$$

Because $\text{Cost}(Q_j^1) < \text{Cost}(Q_j^2)$, replacing y^* at step i is no more costly than replacing x^*.

(II) If the replaced element $z \neq y^*$

Since x is DEAD after step j, x^* in Q_j^1 is equivalent to x in Q_j^2 in terms of store cost. Therefore, to make Q_j^1 the same configuration as Q_j^2, only z in Q_j^1 should be changed to y^*. This takes a cost of (1+U). U is the cost of replacing z and 1 is the cost of fetching y. No load and store are required to change a y to y^*.

$$\begin{aligned} & \text{Cost}(Q_j^1) + (1 + U) \\ = \quad & \text{Cost}(Q_{j-1}^1) + (1 + U) && \text{from (7)} \end{aligned}$$

= $\text{Cost}(Q_{j-1}^2) + (1 + U)$ from (6)

= $\text{Cost}(Q_j^2)$ from (8)

Therefore, if there exists an optimal cost path through Q_j^2, then there will be an optimal path through Q_j^1. Hence, replacing y^* at step i is not worse than replacing x.

(B) x is not in Q_{j-1}^1.

Both x^* and y^* have been replaced during the configuration changes from Q_{i+1} to Q_{j-1}. $\text{Cost}(Q_{j-1}^1) = \text{Cost}(Q_{j-1}^2)$. Since both Q_{j-1}^1 and Q_{j-1}^2 have a miss, and assuming the same element will be replaced, $\text{Cost}(Q_j^1) = \text{Cost}(Q_j^2)$. Hence, replacing y at step i is not worse than replacing x.

We may now state the rules for the generation of configurations for step i+1 from a configuration Q_i when there is a miss.

Rule 1: (This is based on observation 1)
If there exists an x in the DEAD set, then generate only one configuration Q_{i+1}, in which x (select any one x if there are more than one exists) is replaced by I_{i+1},
and exit.

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

Rule 2: (This is based on observations 2 and 3)
If C > D then generate only one configuration Q_{i+1}, in which x_c is replaced by I_{i+1} and exit.

Let C′ = max{ NEXTRD(i,x), x ∈ set 3},
D′ = max{ NEXTRD(i,x), x ∈ set 4}, and
x_d be such that NEXTRD(i, x_d) = C′.

Rule 3: (This is based on observation 4)
If C′ > D′ then generate a configuration Q_{i+1}, in which
x_d is replaced by I_{i+1}.
For each x in Q_i such that x ∈ set 4 and NEXTRD(i, x) > C,
generate a configuration in which x is replaced by I_{i+1}.

3.4. The Algorithm to Find An Optimal Allocation

A configuration, along with its cost and parent information, is called a node. The cost of a node is (as before) $\sum_{j=1}^{i} \text{Cost}(Q_{j-1}, Q_j)$, computed along the path from the root to the current node. We start from an initial configuration Q_0 with cost 0, and generate a set of nodes associated with step 1. 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). while i < n do
    i := i + 1
    for each node N in NODESET(i-1) do
      if Hit then N' = N, insert N' into NODESET(i), parent(N')=N.
      if Miss then generate nodes from N as described in rules 1, 2,
          and 3 as above.
    od
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). Find the node N in NODESET(n) with the least cost.
Trace back through the parent pointers to get the allocation.

In the example in Figure 3.1, Belady's MIN algorithm needs 3 loads and 1 store (the left path). The optimal allocation needs only 2 loads and no stores (the right path). If the above algorithm is used, then only the right branch will be generated.

3.5. Special Cases

If we assume every pseudo-register is read only once, *i.e.*, there are no common sub-expressions [3] the pruning rules can be simplified. Since every pseudo-register is read only once, the DIRTY-LIVE set does not exist, nor does the CLEAN set. A clean pseudo-register exists because it is spilled and reloaded into a real register at its use. However, since it is read only once, it is dead right after its use. Therefore, there are only two sets exist: DEAD and DIRTY-DEAD. The simplified rules are stated as follows:

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

Let D = max{ NEXTRD(i,x), x^* in Q_i }
x_d be such that NEXTRD(i,x_d) = D.

Rule 2:
Generate one configuration Q_{i+1}, in which x_d is replaced by I_{i+1} exit.

The above pruning rules are essentially Belady's algorithm. Due to our definition, a dead register x has an access distance of ∞. Combining the above two rules, the register having the greatest access distance gets replaced.

With the assumption that each pseudo-register is read only once, the program structure is either a tree or a forest. In such a case, a variation of the Sethi-Ullman algorithm [Aho86] can be used to determine the instruction sequence so that a minimal

[3] A local variable or a constant that is allocated in a pseudo-register can be defined as a common sub-expression if it is read more than once.

number of load/store instructions are required.

3.6. A Heuristic Algorithm

The rules that we proposed in the previous section eliminate many unnecessary branches in the WDAG. This makes the algorithm practical for small programs. However, because there are three cases where none of our pruning rules are applicable, complete enumeration is still impractical for large programs. The cases not subject to pruning are: (1) elements in the DIRTY sets (sets 3 and 4) that have a distance greater than that of the most distant clean one; (2) elements in the DIRTY-LIVE set that have a distance shorter than the most distant element in the DIRTY-DEAD set; (3) all elements in the DIRTY-LIVE set. To avoid complete enumeration, we introduce heuristic rules for each case to predict which register might be best for replacement.

Case 1 :

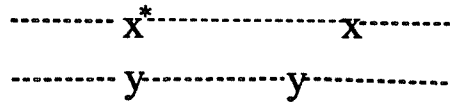


Figure 3.2 Selection between Clean and Dirty Registers

In Figure 3.2, if the distance of $y \cdots y$ is close enough to that of $x^* \cdots x$, then replacing y may be cheaper. This is because the two alternatives may involve the same number of misses, but replacing y needs no stores. Note that replacing the most distant register can reduce misses while replacing a clean register can save stores. Thus a

tradeoff exists between saving stores and reducing misses. The heuristic rule we propose is to use a weighted distance as the selection criterion. For clean pairs (like $y \cdots y$), their distance is multiplied by a weight w_1, $w_1 > 1$. This makes the clean elements more likely to be selected.

Case 2:

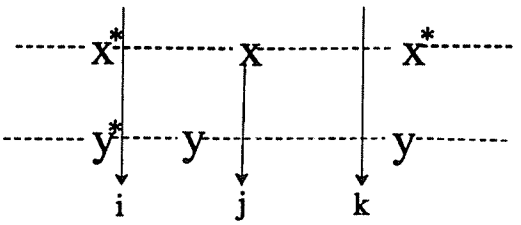


Figure 3.3 Selection between Dirty-Live and Dirty-Dead Variables

With reference to Figure 3.3, replacing $y^* \cdots y \cdots y$ might be better than replacing $x^* \cdots x \cdots x^*$, even when the distance of $x^* \cdots x$ is greater than the distance of $y^* \cdots y$. This is because replacing x at step i needs one store, but replacing it at step k needs no stores. One store is required to replace y, regardless of whether it is replaced at step i or step k. Therefore, if x remains in the register until step j, one store might be saved. This save likely will occur only when the distance of $y^* \cdots y$ is close to that of $x^* \cdots x$. Otherwise, additional misses may require additional loads and stores. In this case, the distance of $x^* \cdots x \cdots x^*$ pairs is multiplied by a weighting factor, w_2, $0 < w_2 < 1$.

Case 3:

For all elements in the DIRTY-LIVE set, access distances are used as the selection criterion.

Heuristic Algorithm WC (Weighted Cost)

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 ∈ set 1 then
         replace x by I_i.
      else
      2.2.2) max := 0
         reg := 0
      foreach x_j in Q_{i-1} do
         case
            x_j ∈ set 2 : distance := NEXTRD(i-1,x_j) * w_1
            x_j ∈ set 3 : distance := NEXTRD(i-1, x_j) * w_2
            x_j ∈ set 4 : distance := NEXTRD(i-1, x_j)
         esac
         if distance > max then
         max := distance, reg := j
      od
      replace x_{reg} by I_i.
od

Examples

We illustrate the algorithm with two examples. To simplify the illustration, we set w_1 statically to 3.0 and w_2 to 0.45 in the following example. In these two particular cases, WC yields optimal bus traffic.

Example 1:

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

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

The replacement process is shown in Table 3.2.

Belady's MIN algorithm produces 4 misses; it needs 3 loads and 1 store. The heuristic algorithm WC also produces 4 misses, but it needs only 2 loads and no stores. MIN can minimize the number of loads only if a write-through policy is used. From the above example, we understand that MIN cannot minimize the number of loads if the write-back policy is used instead.

Example 2:

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

	sequence	r1*	r2	r3	r2*	r1	Results
MIN	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*
WC	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 3.2 Replacement Process of Example 1

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

The replacement process is shown in Table 3.3:

At step 3 there is a miss. If we displace r_1 (as in MIN), eventually the number of load/stores needed will be 5. If we displace r_2 instead (as in WC), 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.

The best values of w_1 and w_2 depend on input programs and the number of available registers. We have investigated a method of finding an appropriate weight for w_1 and w_2 adaptively. The method is to increase w_1 when the miss ratio is getting lower and decrease 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. This adaptive approach requires good initial values for w_1 and w_2, and does not show much improvement over the original algorithm. We therefore believe that

		r1*	r2*	r3	r2	r1	r4*	r5*	r4	r5	r2	r1*	res
MIN	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*
WC	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 3.3 Replacement Process of Example 2

selecting static values would be cost effective. Our experiments suggest that values of w_1 in the range of 1.1 to 1.5 produce good results.

3.7. Existing Heuristic Algorithms

Freiburghouse evaluated four allocation algorithms [Frei74]: MIN, Usage Count, LRU, and LRL (Least-Recently-Loaded). These four algorithms assume write-through (*i.e.* memory always holds correct copies of all registers), and therefore all generate many more stores than necessary. Kim and Tan [Kim78] have extensively studied register replacing problems. Their "life range analysis" is similar to ours. When a live register must be replaced, they pick the most distant one; we call it the Farthest First (FF) algorithm. Fischer presents an algorithm [Fisc87] that tries to eliminate more stores. When replacement of a live register is needed, his algorithm chooses the most distant clean variable first; if there is no clean one left, the most distant dirty one is replaced. We call this the Clean First (CF) algorithm.

3.8. Evaluation

To evaluate the effectiveness of different heuristic algorithms and compare them to the optimal one, we used trace-driven simulation. Two sets of traces are used in the simulation study: a few ordinary C program segments and the Lawrence Livermore Loops [McMa72]. We compiled C programs into their assembly language, and converted a sequence of instructions into a sequence of register reads and writes. Then we used this sequence as input data. To get more pseudo-registers, we assigned as many local variables to registers as possible (using register hints in C). Since our C compiler only takes up to 6 register hint variables, we had to allocate more variables and

temporaries to registers at the assembly level. Usually the basic block we obtain from an assembly program is small. We can, however, intentionally trace some control flow to obtain a larger basic block, as with trace scheduling [Fish81]. Further, in order to have more local variables (so that more symbolic registers will be used), we expanded some procedure calls in-line. This set of traces consists of:

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	Randomly synthesized program segments.
8	A synthesized program segment with in-line expansion and trace scheduling to get larger basic block.

The other set of traces are the first twelve Lawrence Livermore Loops. Since loops 1, 3, 4, 11, and 12 are tight loops, we unrolled them several times to get larger basic blocks. Although loop unrolling is effective, it should not be overused. Many computers have instruction buffers to reduce instruction fetches from memory during loop executions. If the enlarged loop body overflows the instruction buffer, its execution may be slowed down rather than sped up. Therefore, we unrolled the loop several times until the loop size was close to a limit. The assembly code of the loops has been hand optimized so that more common subexpressions can be allocated in registers. In addition, the assembly code programs are run through a code scheduler, which reschedules a program to reduce conceivable interlocks when the target machine is pipelined. Code scheduling typically lengthens the lifetime of pseudo-registers so that more pseudo-registers will be live simultaneously. This will cause more register replacements when the number of registers is small. The simulation results of the two sets of traces are shown in table 3.4 and table 3.5.

Number of loads and stores Generated							
Input Trace	# of regs	LRU	MIN	FF	CF	WC 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 3.4 Evaluation of Heuristic Register Allocation Algorithms

Number of loads and stores Generated							
Input Trace	# of regs	LRU	MIN	FF	CF	WC w1=1.15 w2=0.99	True Optimal
Loop1	2	113	101	96	100	96	96
	4	108	76	60	64	60	-
	8	95	50	19	22	19	19
Loop2	2	86	74	67	69	68	67
	4	76	55	40	43	41	39
	8	61	38	12	14	12	12
Loop3	2	68	58	51	51	50	49
	4	60	43	26	25	25	25
	8	48	27	6	6	6	6
Loop4	2	66	57	48	50	48	48
	4	60	44	29	30	29	28
	8	50	29	11	10	11	10
Loop5	2	70	59	49	51	50	49
	4	62	41	24	23	23	23
	8	41	25	6	5	5	5
Loop6	2	69	58	55	54	53	52
	4	66	43	30	29	31	28
	8	48	23	6	5	5	5
Loop7	2	107	92	84	82	82	82
	4	98	71	55	52	51	51
	8	83	48	17	15	15	15
Loop8	2	246	220	213	217	213	210
	4	237	186	161	164	160	-
	8	216	146	93	97	92	91
Loop9	2	67	61	55	58	55	55
	4	62	50	36	36	36	35
	8	54	34	14	14	14	14
Loop10	2	121	103	91	91	91	91
	4	116	74	53	55	54	-
	8	72	52	29	41	29	-
Loop11	2	68	55	46	46	47	43
	4	53	38	24	27	24	24
	8	42	27	12	12	12	12
Loop12	2	66	59	57	58	56	56
	4	64	47	40	42	38	-
	8	55	29	14	13	14	13

Table 3.5 Evaluation of Heristic Register Allocation Algorithms (for Livermore Loops)

When w_1 is equal to 1, our heuristic is close to the FF algorithm; when w_1 is a very large number it degenerates into the CF algorithm. Thus, the Weighted Cost (WC) algorithm will always have an intermediate result between that of FF and that of CF when only w_1 is used. With a properly selected w_1, the WC result usually is close to the lower of the two (FF and CF), as shown in Table 3.4 and Table 3.5. In addition, with the help of w_2, WC will sometimes outperform both FF and CF.

Since LRU and MIN, which were used to evaluate register allocation [Frei74], assume that memory always holds correct copies of all registers, these algorithms generate many more stores, as shown in Table 3.4 and Table 3.5. Therefore, we will exclude LRU and MIN in the following discussions.

Table 3.4 shows that for small basic blocks with small numbers of variables (trace 1,2,3), the three algorithms (FF, CF, and WC) perform identically to the optimal algorithm. This means, for the small basic blocks which are common in C programs, a heuristic algorithm like FF or CF is good enough. For larger basic blocks with more variables (trace 5,8), the heuristic algorithms produce slightly more loads and stores than the optimal one. Among the heuristic algorithms, FF sometimes outperforms CF and vice-versa, while WC always has intermediate results close to the lower.

Table 3.5 represents results from large basic blocks. Without our pruning rules, it is computationally infeasible to obtain an optimal allocation of registers for large basic blocks. Although our optimal algorithm also failed to obtain an optimal allocation of registers in a few cases, due to insufficient memory space, it did terminate in most cases, allowing us to make comparisons with the heuristic algorithms. Table 3.4 shows that heuristic algorithms have near-optimal results even when basic blocks are large, but with more observable difference when compared with the results of small basic

blocks. This confirms that heuristic algorithms being studied are effective for register allocation for large basic blocks. Our heuristic algorithm (WC) has slightly higher complexity in implementation (less than 20 lines of C code) but usually outperforms both FF and CF. Therefore, we believe the WC algorithm is a good candidate to be considered when allocating registers for large basic blocks.

3.9. Replacement vs Graph Coloring

The well-known graph coloring allocation [Chai81, Chai82] can be applied to local register allocation. In the graph coloring model, register spilling refers to storing a pseudo-register into memory and reloading it at a later time. Spilling is synonymous with "replacing a live register" in this chapter. Graph coloring allocation is an elegant approach if no spilling is required. But once the coloring process is blocked due to insufficient colors (registers), trouble begins. The spilling process is slow, and it is hard to decide (1) which node to spill; (2) how many nodes must be spilled; and (3) where to insert spilling code. Chaitin claims spilling is not a problem in their implementation, because (1) spilling occurs rarely; and (2) spilling rapidly reduces the interference graph to be colorable. This is true for small basic blocks (or small procedures if it is applied to global register allocation). However, if we want to use optimizing techniques like in-line expansion and loop unrolling, then the increase in local data items will complicate the interference graph and make the spilling a hard problem.

In graph coloring allocation, if the interference graph of a straight-line program is N-colorable, then the 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 allocated pseudo-register 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 Chaitin [Chai82] is similar to a reference count algorithm [Aho86, Powe84] which does not adequately consider clustered accesses. Therefore, the replacement based algorithm is superior to the graph coloring algorithm in straight-line programs. It is unfair, however, only to make comparisons for straight-line programs, since graph coloring allocation is designed primarily for global allocation. Nevertheless, the replacement based algorithm can be extended to global allocation, and the extension can take advantage of software branch predictions [Fish81]. The extension to global allocation will be described in the next section.

Register replacing is similar to the replacement problem in a paging system or in a cache. The difference is that a compiler can use look-ahead techniques to exploit future information at compile time. By using the future information, register replacement can be handled efficiently. The graph coloring model, which is often handicapped by the spilling problem, depends on either a large number of registers or small procedures to avoid spilling. Since typical procedures are small, as are basic blocks, register spilling is unlikely to occur. However, this does not mean that we should encourage the use of small procedures and small basic blocks. There is a cost associated with each state transition from one procedure or basic block to another. For procedures, that cost is register saving and restoring. For basic blocks, that cost is initial loads at block entry and final stores at block exit. The smaller the unit, the higher the transition overhead. Therefore, larger units give better opportunity for more efficient allocation. In addition,

recent research in pipelined and parallel processing require large basic blocks to supply sufficient fine-grained parallelism [Fish81, Elli85]. As the size of basic blocks becomes larger, the problem of good register replacement becomes more important.

3.10. The Reevaluation of the Perfect Allocation Model

Since we have already studied algorithms for minimizing bus traffic, the "perfect" register allocation model can be reevaluated. The heuristic algorithm (WC) is used rather than the optimal algorithm to gauge ideal performance because (1) the optimal algorithm requires exponential time and space; it is infeasible to be used to evaluate a trace with millions of steps, and (2) the algorithm WC has near-optimal performance even with large basic blocks.

Comparison of Bus Traffic (Average of five traces)		
no. of words	Cache (LRU)	Registers (Perfect Allocation)
2	55271	40294
4	36022	22534
8	23716	10563
16	6183	1987
32	1201	70
64	187	12

Table 3.6 Bus Traffic of Perfect Register Allocation and Data Cache

Table 3.6 shows that using registers has the potential for less bus traffic than a data cache, especially when the registers have an effective size of 32 or larger. It is hard to conclude how much difference this makes in improving the execution speed. However, if the off-chip data bandwidth is the performance bottleneck and the stack access is the major part of the data bandwidth, then halving the stack access traffic might nearly double the performance. The reader may argue that increasing the cache size may be

more effective in reducing the bus traffic. But for a limited on-chip area, registers can be more effective.

Although it is difficult to attain performance approaching that of ideal allocation, compiler techniques are improving; anti-alias analysis has been developed and used [Nich85, CoutB86]; new programming languages such as Euclid [Pope77] and Ada [Ledg81] are designed to restrict uncontrolled aliases; and research on software branch prediction is being conducted [Elli85, Fish81, HsuP86]. We therefore believe that using registers to reduce off-chip memory accesses for the *stack* has a promising future.

Some computer architects favor architectures which simplify the optimization work of a compiler. However, the successful development of portable optimizers [Chow83, Ausl82] reduces the cost and effort of the optimization work. In addition, we believe that in order to use a limited storage effectively, sophisticated allocation is necessary. For example, if the CRISP (C machine) wants to use its limited top-of-stack area effectively, its compiler should minimize the frame size of each procedure so that more frames can be captured in the top area. (In fact, this is done in one of the CRISP compilers [Band87]) Also, if a frame size is larger than the top-of-stack area, then it is the compiler's responsibility to allocate frequently used variables on the top of the frame to speed up their access.

3.11. Global Register Allocation

In previous sections, 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 temporaries or 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. Since programs spend most of their time in inner loops, most of the global allocation algorithms pay attention to allocating registers in inner loops.

3.11.1. Previous Work

A simple yet effective algorithm, known as reference count [Aho86], has been used in the Fortran H compiler [Lowr69] and a Modula II compiler [Powe84]. This algorithm identifies frequently used variables, constants, and base addresses by counting their static references, and allocates to registers as many as possible.

Day [Day70] formulates global register allocation as an integer programming problem. Branch and bound algorithms which lead to optimal solutions are given. Heuristic algorithms are also provided. One heuristic algorithm is similar to the reference count algorithm. Interferences among variables are analyzed so that more than one data item can be allocated to one register. Day's algorithms are based on the IBM/360 model, for which data items can be accessed directly from storage. This is different from our model which is based on a Load/Store architecture.

Beatty [Beat74] 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.

Harrison [Harr75] 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 have priority to be allocated in registers.

Wulf, *et al.* [Wulf75] separate register allocation from register assignment in the Bliss-11 compiler. Temporary names (TN) are assigned to selected variables, common subexpressions, and loop invariants in the TN phase and are mapped 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. All variables are initially allocated in registers. Then, lifetime analysis is used 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.*, [Chai81, Chai82] 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 [Yers71], Cocke [Alle76], and others, it was successfully developed and implemented by Chaitin. The global register allocation 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 two quantities interfere with each other, that is, if they are simultaneously live at some

point in the program. The problem is to assign different colors (registers) to connected nodes. It is hard to obtain an optimal coloring, but the implementation showed a fast heuristic method for assigning colors to these graphs generally resulted in a very good assignment. 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 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 is 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 (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). Despite their delicate spilling process, Chow and Hennessy's implementation of the live range of variables restricts the sharing of registers. A live range of a variable is an isolated and contiguous group of nodes in the control flow graph in which the variable is defined and referenced. Each variable in a procedure is assumed one live range even though it may be defined multiple times. Assume two frequently used variables A and B, each is defined several times but both are never live simultaneously. Based on Chaitin's model, A and B can share the same register. In Chow's model, since A and B both have high priorities, their live ranges will be assigned to different registers (colors). Although a live range can be split into several smaller live ranges, spliting only occurs when the colors are used up. When the colors are used up, A and B have already been allocated and their live ranges will not be split.

3.11.2. Important Considerations for Good Allocations

In addition to the common principle that frequently used data items should be allocated in registers, there are some important criteria in designing good register allocation algorithms. From the previous work, the criteria can be listed as follows: (1) variables (including temporaries) with disjoint lifetime should share registers; (2) the information of variable access clustering is important in handling spilling; (3) loads and stores (spilled code) should be placed in less frequently used blocks. We will show that extending our local allocation scheme to global allocation fulfills the above criteria.

3.11.3. 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. One compiler attempted "hot spot" optimization of this sort: the original Fortran I compiler, which used branch frequency information in its optimization phase [Back57]. 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.

3.11.4. The Extension of Local Allocation to Global Allocation

The extension of our local allocation algorithm to global allocation is based on the approach used by the Fortran I compiler. A brief outline of this extension is as follows. Using estimated branch frequency, regions are formed so that registers can be assigned to the most frequently executed areas first, then to the next most frequently executed area, and so on, until the entire program has been treated. Each region is a path (or a trace [Fish81]) which can be treated as a basic block. When a path has been processed, the register configurations at each split and rejoin are recorded (see Figure 3.5) in order to maintain consistency with the processing of other paths. A similar process has been applied in different studies [Back57, Kenn72, Harr75, Mosh85, Kran86]. Figure 3.4 serves to illustrate this idea. Implementation detail of trace scheduling can be found in [Elli85]. In Figure 3.4(a), a flow diagram shows a loop with an IF-THEN-ELSE structure inside its body. Suppose the path A-B-D is taken more frequently than the path A-C-D. Then the register allocator will treat A-B-D as a basic block, and allocate

registers accordingly. Register configurations are recorded at the split and rejoint as shown in Figure 3.4(b). Then the allocator performs register allocation for block C. Load and store instructions are inserted around block C to maintain the consistency with the register configurations at the split and rejoint. The approach is to optimize a more frequently used path while letting a less frequently used path pay an extra price, a principle that has been used widely.

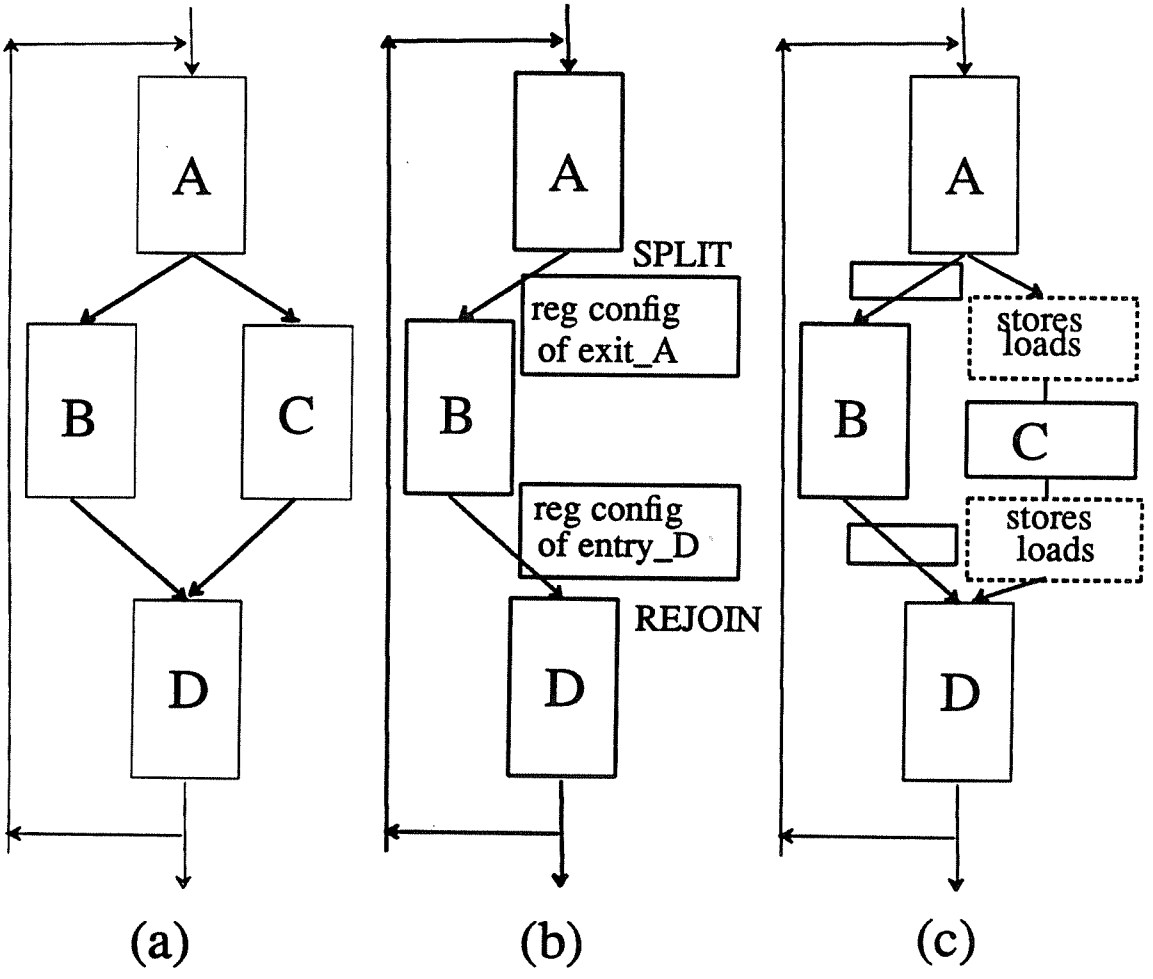


Figure 3.4 Extension to Global Register Allocation

Our replacement-based allocation algorithm keeps frequently used data items in registers (it works like a perfect cache); reuses registers for variables having disjoint lifetime (it replaces dead registers first); and takes variable access clustering into account in spilling registers (which register to spill is based on its access distance). Combined with trace scheduling techniques, the replacement-based allocation algorithm will place the spilled code (loads/stores) in a less frequently used area. Therefore, this approach is designed to fulfill the important criteria stated before.

3.11.5. Some Required Modifications

Some modifications are necessary for the above extension.

(1) In local allocation, if a variable (a pseudo-register) is dead, then no stores are needed to update its value in memory when it is spilled. This is a little tricky in global allocation, since a variable may be dead in the current trace but live on a split trace. In order to generate correct code, a store must be inserted if the spilled variable may be live and dirty. Therefore, global flow analysis should be used to tell whether a variable is dead or not — it is dead at a point if its current value will never be used again — rather than only looking ahead on the current trace.

(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 unrolled once, and allocation carried out in normal fashion through the first of the two copies, with look-ahead extending into the second copy. The content of registers on exit from the first copy thus determined was now applied as the initial condition.

(3) It is easy to do allocation for the first selected trace. But for later selected traces, the allocator needs to maintain the consistency at the rejoins. It needs some care to avoid generating too many redundant register movements.

(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] can be applied for such a purpose.

3.12. Register Allocation of Global Variables

While we have concentrated primarily on accesses to local variables up until now, many programs make extensive use of global variables, and these must be supported efficiently as well. From Ditzel's study one might conclude that global variables are accessed infrequently [Ditz82]. However, Ditzel's study is largely based on system applications; there are clearly many applications (e.g. scientific computing) where many, or even most, variables are global. There is strong evidence that a cache is often not effective for global variables [Axel83]. From our study reported in chapter 2 we concluded that major temporal locality in global accesses comes from a small number of scalar variables. Because they are accessed frequently, these scalar variables are obvious candidates for registers. However, many of these references are complicated by aliasing problems — being passed by reference parameters to procedures, for example — and therefore cannot be allocated to registers. In-line expansion overcomes this restriction by reducing the use of aliases.

The remaining global accesses exhibit little temporal locality and therefore cannot be supported well by any on-chip memory relying primarily on temporal locality. Spatial locality is exploited in a cache to raise the hit ratio so that memory latency can

be hidden. Caches that bring in large blocks work if they have sufficient bandwidth, but are wasteful in the use of the available processor/memory bandwidth. On-chip cache must be designed to minimize this bandwidth [Good83] and therefore is unlikely to perform well. Not only will unneeded data be fetched, but anticipation of requests is very difficult.

The use of registers can potentially overcome both of these constraints. The program can preload data into registers when it has advance knowledge of their use. The one seeming advantage that a cache has is that more than a single word can be requested and transferred at a time. We note that the same mechanisms can be employed for registers. In fact, registers are more flexible because the words fetched need not be contiguous. The fetched data then can be placed in vector registers, queues or other structures visible to the programmer. Preloading fetches exactly the data that is needed, therefore unlike the cache, no additional traffic is introduced. The task of preloading is typically performed by code scheduling– rearranging code sequence at compile time–so that load instructions can be overlapped with other operations. Ideally, preloading can achieve 100% hit rate without adding more traffic.

Code scheduling detects the parallelism in a program, arranges independent instructions to be interleave so that latency can be hidden (both memory latency and functional unit latency). There are two problems associated with code scheduling: (1) it is not very effective if the basic blocks are small, since small basic blocks have very little parallelism; and (2) it often complicates the task of register allocation. In the next chapter, we discuss the issues of large basic blocks and compiler techniques to generate large basic blocks. In chapters 5 and 6, the interdependency of register allocation and code scheduling is studied and some solutions are proposed.

Chapter 4
Compiler Techniques for Increasing Basic Block Size

A basic block is a sequence of consecutive instructions which may be entered only at the beginning, and when entered are executed in sequence without branches (except at the end of the basic block). It is the basic unit for local optimizations, including code scheduling. Typically, basic blocks are very small. For example, Clark [Clar84] found that 25 percent of the VAX instructions executed are branches. This means that the average size of a dynamic basic block is only three instructions long. Although for RISC type architectures [Patt85], branch frequency tends to be lower, branches still constitute a high portion of the instructions executed [Henn84]. For pipelined machines, branches not only cause high penalties in execution but also severely limit the usable parallelism due to small basic blocks. Eliminating branches and generating large basic blocks, therefore, is of great importance in making effective use of pipelined machines. Some compiler techniques for generating large basic blocks are discussed in this section.

4.1. Reducing Conditional Jumps

If it can be determined at compile time that a conditional branch will always take just one of the possible paths, then it can be removed, along with the path which is never taken. Constant propagation [Wegm84] and copy propagation [Aho86] are two ways to eliminate such unnecessary branches.

Short-circuit code [Logo81] is often generated for boolean operations in many compilers. It has the advantage that fewer instructions will be executed since some expression evaluations can be bypassed. However, short-circuit code uses many

conditional branches, resulting in many tiny basic blocks. In order to obtain larger basic blocks, short-circuit code should be avoided if the language definition allows.

With appropriate architectural support, the number of conditional branches can be further reduced. For example, the **max** instruction, which takes two operands and returns the bigger one, can be used to implement a C statement like a=(b>c?b:c) without using a conditional branch.. Although it is difficult for the compiler to use this feature in code generation, the programming language can be augmented with function calls. When programmers use the **max** function call, the translation is quite straight forward. Ellis [Elli85] suggested a more generalized machine instruction, **select**,

select(b, x1, x2)

which returns either **x1** or **x2** depending on whether **b** is true or false, to avoid unnecessary conditional branches.

4.2. Code Duplication

Code duplication is a common way to trade code space for more parallelism. Many techniques belong to this category. They include loop unrolling [Dong79], code replication [HsuP86], trace scheduling [Fish81], in-line expansion [Sche77], unswitching [Alle72], and multiversion program transformation.

4.2.1. Loop Unrolling

Technique and advantages

When a loop is unrolled, its body is replicated so that the calculations performed in several iterations of the unmodified loop are performed in one iteration of the

unrolled loop. In Figure 4.1, a Fortran DO loop is unrolled four times. The basic block in the unrolled loop is four times larger than before. Loop unrolling reduces the number of branches that must be performed, and increases the oppertunitsies for more local optimizations. Furthermore, calculations from several iterations can often be overlapped, increasing parallelism.

Loop unrolling is also helpful for register allocation. In Figure 4.2, after loop unrolling, Y(I) can be identified as a common subexpression and thus be allocated in a register, saving store and load instructions.

```
   DO 10 I = 1,N
            Y(I) = Y(I) + A*X(I)
10 CONTINUE
```

(a) original loop

```
   M = N - MOD(N,4)
   DO 10 I = 1,M,4
            Y(I) = Y(I) + A*X(I)
            Y(I+1) = Y(I+1) + A*X(I+1)
            Y(I+2) = Y(I+2) + A*X(I+2)
            Y(I+3) = Y(I+3) + A*X(I+3)
10 CONTINUE

   DO 11 I = M+1, N
            Y(I) = Y(I) + A*X(I)
11 CONTINUE
```

(b) unrolled loop

Figure 4.1 Unrolling A Fortran DO Loop

```
      DO 10 I = 1,100
            Y(I) = Y(I-1) + X(I)
10 CONTINUE
```

(a) original loop

```
      DO 10 I = 1,100,2
            Y(I) = Y(I-1) + X(I)
            Y(I+1) = Y(I) + X(I+1)
10 CONTINUE
```

(b) unrolled loop

Figure 4.2 Loop Unrolling and Register Allocation

Limitations

Loop unrolling is not as effective for WHILE loops as for DO loops. In the unrolled WHILE loop, neither the number of branches will be reduced nor the size of the basic block will be increased as shown in Figure 4.3. However, if we are confident the WHILE loop will iterate many times, unrolling it gives an opportunity to apply trace scheduling optimizations. We will describe this further in the section on trace scheduling.

One important constraint on loop unrolling is the presence of "instruction buffers" on most advanced computers. If the unrolled loop overflows the instruction buffer (or instruction cache), then increased instruction fetches may slow down execution. However, it is well-known that instruction prefetch is very effective on straight-line programs. Therefore, via prefetching, the instruction fetch delays caused by unrolled loops may be negligible. However, this argument assumes extremely high memory bandwidth so that instruction prefetching will not interfere with data fetching. Since

```
While (A(I) <= N)
        Body;
```

(a) original loop

```
L10:    IF (A(I) > N) EXIT
        Body;
        IF (A(I) > N) EXIT
        Body;
        :
        :
        :
        IF (A(I) > N) EXIT
        Body;
        Go To L10;
```

(b) unrolled loop

Figure 4.3 Unrolling A WHILE Loop

limited memory bandwidth is the major performance bottleneck of modern single-chip processors, the unrolling depth must be a major concern. A smart optimizing compiler would take the size of the instruction buffer and the number of pipeline stages into account to determine how many times a loop should be unrolled.

4.2.2. In-Line Expansion

Technique and advantages

In-line expansion (also called procedure integration) is a program transformation that replaces a call to a procedure by the body of that procedure. This transformation eliminates the overhead of the subroutine calls: control linkage, registers saving/restoring, and parameter passing. It allows better global optimizations, because

more constants can be propagated, more common subexpressions can be found, and better register allocation can be done. It also makes the data dependency analysis more exact, since only the effects of the one call must be taken into account. In-line expansion is very effective in increasing the size of basic blocks. First, it eliminates two branches (a call and a return). Second, since the actual parameters are often constants, by propagating the constant parameters some conditional branches can be eliminated at compile time, producing larger basic blocks.

Limitations

(1) Recursive procedures. Recursive functions or procedures cannot be eliminated by in-line expansion. However, recursive procedures are not used very often. About 12% of PASCAL procedures are recursive according to Madhavji's study [Madh82]. In addition, recursive procedures can be transformed into nonrecursive or iterative routines. Although a recursive procedure can not be entirely eliminated, expanding it several times is still helpful in assisting optimizations. For example, expanding the Ackermann's function a small number of times can produce a significant speed up. The C++ compiler expands some recursive procedures once.

(2) Procedure environments. In block structured languages, procedures are usually associated with complex environment. In-line expansion of those procedures must handle the associated environments correctly. In the VAXELN Pascal compiler [Macl84], formal procedures and nested procedures are not expanded in-line.

(3) Separate compilations. Without the procedure body, in-line expansion can not be performed. One distinction between debug runs and compilation runs should be

made here. In typical programming environments, a program is broken into many small compilation units because speed of compilation is important. After the program is debugged, the compilation units can be merged into a few large ones.

(4) Code space restriction. One major concern is that in-line expansion may significantly increase the code space when a procedure is called many times statically. In theory, the code space could be exponentially increased by in-line expansion if every procedure calls more than one procedure and is called more than once. Especially when a procedure is called several times in a loop, its expansion may cause the loop body overflow the instruction buffer. However, in practice, many procedures are used on the purpose of clarity, modularity and data abstraction. These procedures are simple, small and often called only once. For example, no code space explosion have been experience by Scheifler [Sche77] and MacLaren [MacL84]. How to expand procedures selectively is of great importance. The PL.8 compiler expands leaf procedures and simple functions because their bodies are relatively small compared to the calling overhead. The Standford UOPT includes a cost-driven in-line expansion phase [Chow85]. P. Hsu [Hsu86] suggested limiting in-line expansion only to those execution paths that having a high probability of being taken. Ball [Ball79] described a technique for predicting the code improvement that can be expected due to constant folding and test elimination when a procedure call involving constant actual parameters is expanded in-line. A precise prediction of which procedures should be expanded would be extremely hard to obtain. However, a simple guideline can be given: do not expand a procedure which is relatively large, has few constant parameters, and is called many times statically.

4.2.3. Code Replication

Technique and advantages

In order to make the Decision Tree Scheduling (DTS) algorithm more effective, P. Hsu suggested one technique called code replication [HsuP86]. As shown in the Figure 4.4, two if-then-else structures can be merged into a large decision tree. The joint block S5 is replicated in each leaf of the decision tree, thus increasing the size of the basic blocks of the leaves. The size of basic blocks can be further enlarged when code replication is used with multi-way branches [Fish80, Karp85]. For example, in Figure 4.5, which assumes the two conditions in Figure 4.4 are independent, the use of multi-way branch can produce larger basic blocks.

Limitations

Code replication increase code space rapidly. Therefore, P. Hsu used this technique primarily for those frequently used branches in the decision tree.

4.2.4. Trace Scheduling

Technique and advantages

Trace scheduling [Fish81] is a technique for code compaction across basic block boundaries. There are four phases in the scheduling process. First dynamic information concerning the flow of control of the program is obtained and is used at compile time to perform "software branch prediction", the prediction of which paths are more likely to be executed. A selected path is called a "trace". Once a trace has been selected for

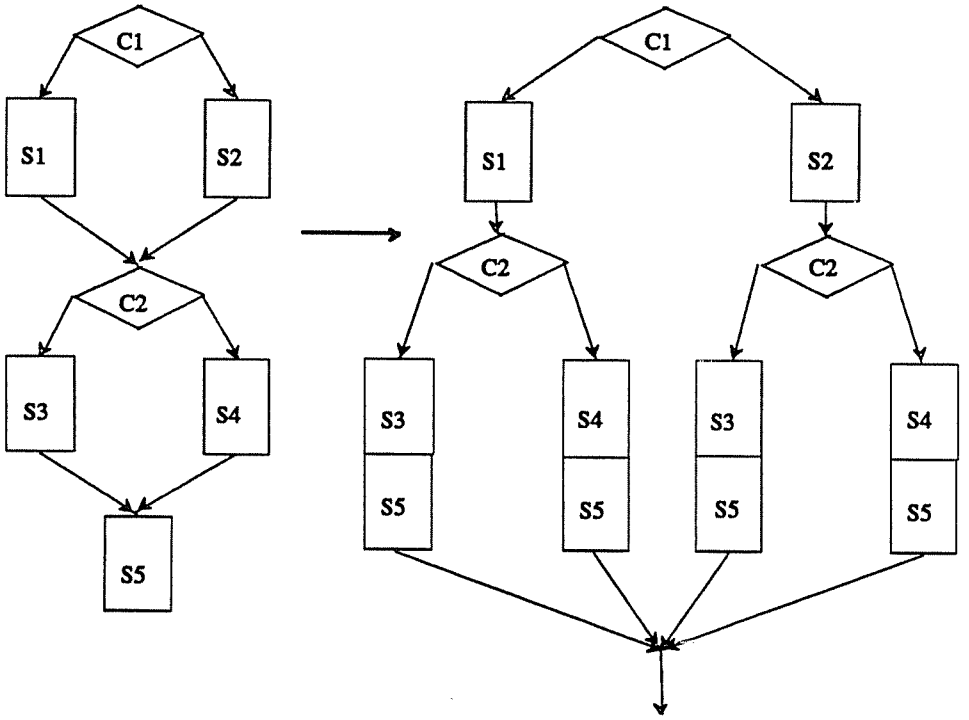


Figure 4.4 Code Replication

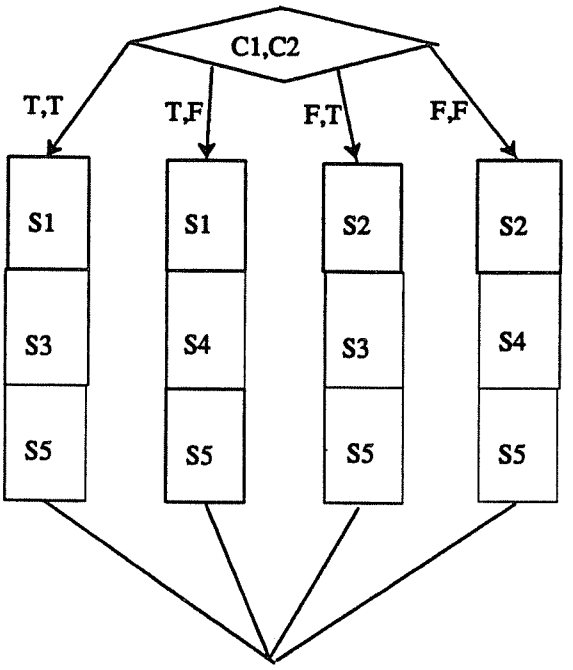


Figure 4.5 Code Replication with Multi-way Branch

compaction, some preprocessing is performed on it so as to disallow code motion of operations that would alter variables which are live off the trace. Then the scheduler is allowed to compact the entire trace as if it were one large basic block. After compaction, the scheduler will have made many code motions across conditional jumps. These code motions may have altered the state of the program with respect to jumps into or out of the trace. To restore the correctness of the program, new code is inserted at the trace exits and entrances to recover the correct machine state.

The example in Figure 4.6 illustrates the process of trace scheduling. Four basic blocks which represent an if-then-else structure are shown in the example. Suppose the block sequence 1, 2, and 4 is the selected trace. The instruction E=3 is not allowed to move from block 2 to block 1 in the preprocessing stage since it will alter variable E

which is live in block 3 and which is on the off trace. The scheduler can move instruction A=B+C from block 1 to block 2, or move instruction T=D from block 2 to block 1 (assume T is not live in block 3). If the instruction A=B+C is moved from block 1 to block 2, then a replication of this instruction should be inserted in the recovery block between block 1 and block 3.

From the example in Figure 4.6, it is obvious that a large basic block can be created just by moving instructions from block 1 to block 2. The expense is that the same number of instructions should be replicated in block 3. This transformation has the same flavor as code replication.

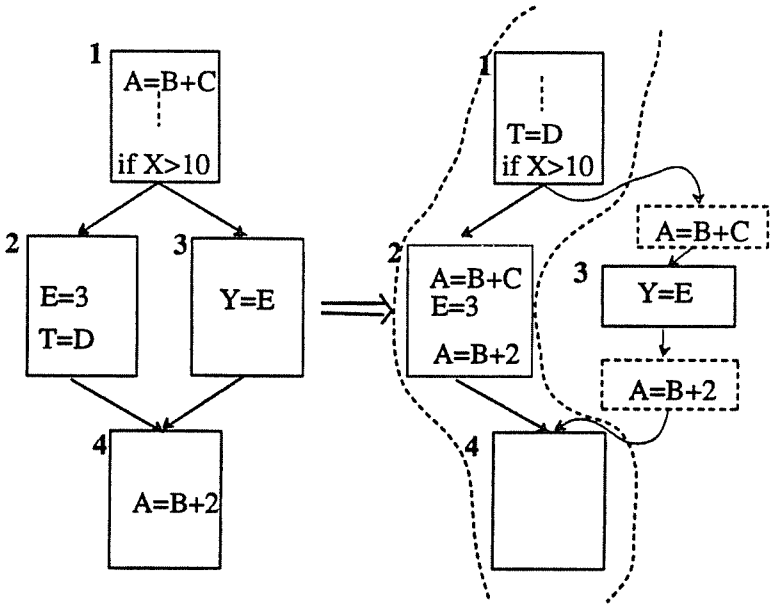


Figure 4.6 Trace Scheduling

Trace scheduling was originally designed for code compaction. But it is also useful for code scheduling. Both code compaction and code scheduling require large basic blocks to be effectively used.

Trace scheduling is most effective when used with loop unrolling. In the section on loop unrolling, it was mentioned that unrolling WHILE loops is not as effective as unrolling DO loops. However, with trace scheduling, all conditional jumps in the unrolled loop can be predicted as "fall through" (not taken branches) as long as the loop will be executed many times. The combination of loop unrolling and trace scheduling creates a notion of a *pseudo basic block*, a long section of code, with a single entry and multiple exits, all of which except the last one are unlikely to be taken. A pseudo basic block can be treated as a basic block as long as some care is taken during postprocessing to maintain the correctness of the program. Some architectures make this especially easy (e.g. WISQ [Ples87]). With loop unrolling, it is easy to obtain large pseudo basic blocks.

Limitations

The success of trace scheduling heavily depends on the correct prediction of branches. For scientific applications, most branches are highly predictable, and so trace scheduling is quite effective. Some studies [Ditz87] have reported that for non-scientific computation, branches can also be predicted with a high success rate. If this is true, then trace scheduling may also be successfully applied to non-scientific applications.

Trace scheduling is restricted to some extent by instructions which cannot be moved across basic blocks, as with the instruction E=3 of block 2 in the example. However, these instructions could be moved across basic blocks as long as we provide a

way to undo them when the branch prediction turns to be wrong. The WISQ [Ples87] architecture, for example, has a feature to nullify instructions easily, thus making trace scheduling more effective.

Although Nicolau has shown that trace scheduling does indeed terminate [Nico85], trace scheduling can lead to code space explosion [Elli85]. However, for scientific code with simple control structures and few cascaded conditionals, code explosion can be controlled.

Trace scheduling is hard: it is hard to implement, it may need programmers to specify conditional jump probabilities and assertions for memory-reference. Furthermore, it takes lots of time to compile.

4.2.5. Unswitching and Multi-version

Technique and advantages

Unswitching [Alle72] is a technique which hoists loop-invariant branch tests outside the loop and replicates the loop body so that once the execution falls into the correct replicated loop, there is no more intra-loop branching. Figure 4.7 illustrates such an approach. The loop body will be doubled if there is one loop invariant test, quadrupled if there are two independent loop invariant tests, and so on. This program transformation will make a loop execute more efficiently than before in a pipelined machine. This idea can be extended to a more general approach. In many programs, it is common to define some flags at the beginning of the execution and test the flags repeatedly. Since many flags are not changed during program execution, these tests are unnecessary. One possible transformation to reduce redundant tests is to replicate the

program several times so that there are several versions, each of which corresponds to a particular set of test values. The invariant tests are checked at the start of execution, and then the corresponding version is selected. All of the invariant tests in each version can be removed through constant propagation. Figure 4.8 shows an example of the multiversion approach.

Unswitching and multiversion approach expands primary static code size but not necessarily dynamic code size. With decreasing memory cost and increasing main memory size, static code size is less dominant than it was a decade ago. **Limitations**

Unswitching and multi-version transformation increase code space drastically. Without proper control, it may cause code space explosion. The execution frequency estimate of loops and the number of tests that can be hoisted should be taken into account to determine whether the above transformation is profitable.

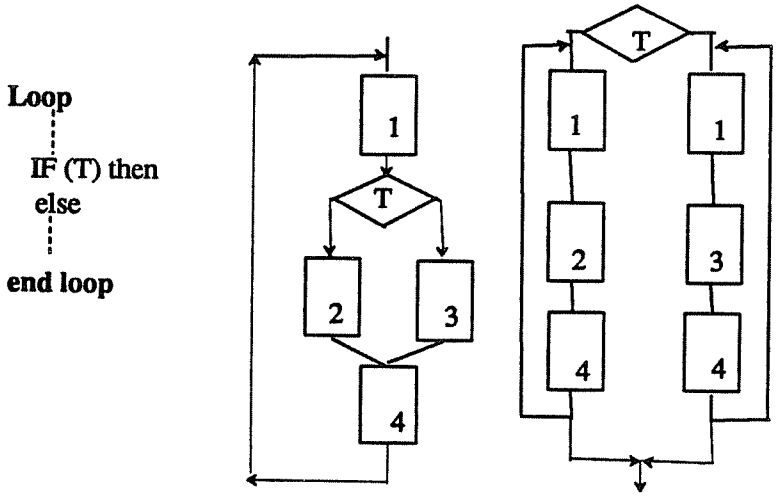


Figure 4.7 Unswitching

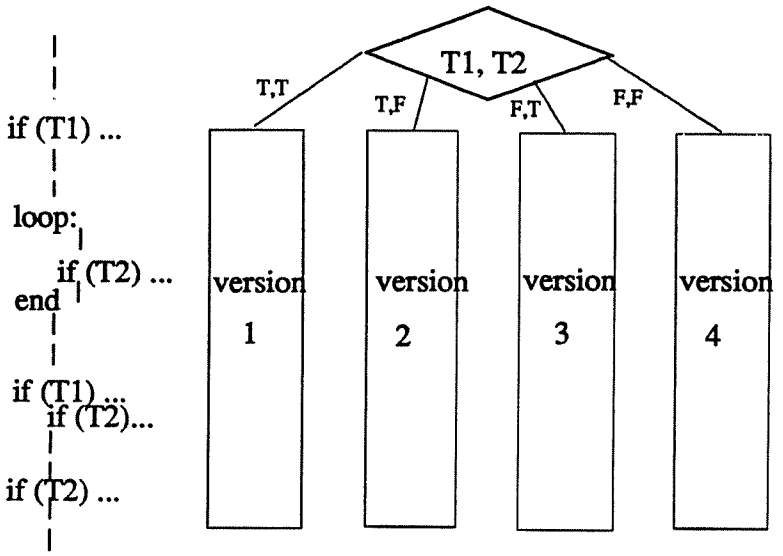


Figure 4.8 Multi-version

4.3. Others

4.3.1. Loop Fusion

Loop fusion (also called loop jamming) is a conventional compiler optimization that transforms two adjacent loops into a single loop. For example, the loops

would be fused as:

```
   DO 10 I = 1,N
            A(I) = B(I)
10 CONTINUE

   DO 20 I = 1,N
            C(I) = D(I) + E(I)
20 CONTINUE
```

Figure 4.9 Loops Before Loop Fusion

```
   DO 10 I = 1,N
            A(I) = B(I)
            C(I) = D(I) + E(I)
10 CONTINUE
```

Figure 4.10 Loop After Loop Fusion

Fusion helps to reduce the overhead of loops. More importantly, the basic block size is increased. Several limitations should be considered:

(1) Sometimes, loop fusion is not legal. For example, the following loops cannot be fused, since the original data dependency will be changed if the two loops are fused.

```
   DO 10 I = 2,N
            A(I) = B(I)
10 CONTINUE

   DO 20 I = 2,N
            C(I) = A(I+1)
20 CONTINUE
```

(2) Loop fusion may decrease spatial locality. For example, the loops in Figure 4.9 may be very efficient since the loops can take advantages of the high spatial locality of array B. The fused loop in Figure 4.10 may be less efficient due to competing accesses of arrays B, D, and E.

(3) The increased loop size may overflow the instruction buffer.

Chapter 5
Code Scheduling with Limited Register Spilling

5.1. Introduction

Pipelining is a common technique used in high-performance computers [Kogg81]. It increases system performance by overlapping instruction execution. Ideally, more pipelined stages (*i.e.* a more finely segmented pipeline) means higher throughput. However, the presence of branch instructions and inter-instruction data dependencies often restricts the effectiveness of a long pipeline. Hardware techniques like issuing instructions out of order and internal forwarding [Toma67, Thor70] have been used occationally to alleviate the data dependency problem. However, they are not used widely because (1) they are expensive; (2) their complexity may slow down the clock rate. On the other hand, code scheduling, a software technique that rearranges the code sequence at compile time to reduce possible run-time delays, has been shown to be effective for improving the performance of pipelined processors [Henn83, Youn85, Gibb86].

Code scheduling works because parallelism exists in basic blocks. The code scheduler can interleave several independent instruction sequences so that the hazards can be largely eliminated. However, the observed parallelism in typical basic blocks is often limited to a factor of two to three [Tjad70]. This implies that pipelines with more than three stages may be ineffective. This is probably one reason why most commercial microcomputers do not have many pipelined stages in their implementation. Vector processors [Hwan84] often have long pipelines, since vector operations consist of long sequences of operations on different array elements with no

inter-operation data dependencies. Large basic blocks allow code schedulers to find and exploit more parallelism within programs [Elli85]. Compiler techniques to generate large basic blocks have been discussed in the previous chapter.

While code scheduling is effective in reducing pipeline interlocks and hiding memory latency, it creates a problem for register allocation. Code scheduling increases the time between a write to a register and the reads after the write. Having longer register lifetimes increases the number of simultaneously live registers, interfering with register allocation. Code scheduling may cause the register allocator to spill some registers. This is a major reason why researchers choose postpass code scheduling — code scheduling after the register allocation is done [Henn83, Gibb86] — rather than prepass scheduling. However, since the register allocator may inadvertently introduce dependencies by allocating the same register for unrelated instructions, postpass code scheduling is more restricted than prepass code scheduling. When basic blocks are small, this restriction makes little difference. But for large basic blocks and long pipelines, it may make a significant difference in performance.

We introduce two approaches to attack the interdependency problem between code scheduling and register allocation. This chapter describes a code scheduling technique which combines two code motion techniques — one to reduce pipeline delays and the other to minimize register usage— into a single phase. By keeping track of the number of available registers, the scheduler can choose the appropriate scheduling technique to schedule a better code sequence. This technique also considers spilling as a trade-off with runtime interlocks because for some heavily pipelined processors, spilling may not be more expensive than long interlocks. The other approach is based on a DAG-driven register allocator which uses a Dependency DAG

to assist in assigning registers; it introduces much less extra dependency than does an ordinary register allocator. It will be described in the chapter 6.

5.2. Background

5.2.1. Code Scheduling Constraints: The Dependency DAG

Code scheduling algorithms generally reorder instructions to improve program execution time [Henn83, Gibb86]. The ordering must preserve the original partial order imposed by operational precedence constraints. DAGs (Directed Acyclic Graphs) are normally used to represent program precedence constraints [Aho86]. A DAG defines legal evaluation orders within a basic block; nodes represent instructions, and edges represent serialization dependencies between instructions. An edge leading from instruction *A* to instruction *B* indicates that *A* must be executed before *B* in the scheduled code sequence. An example code sequence and its dependency DAG are shown in Figure 5.1. This example program will be used repeatedly in this chapter.

5.2.2. The Use of Low Level Intermediate Languages

The use of intermediate languages (IL) simplifies the code generation and optimizations. The PL.8 compiler [Ausl82] uses a low level IL with an unlimited number of symbolic registers. In the register allocation phase, the symbolic registers are mapped into a limited number of physical registers. The IL used in this paper is very similar to the assembly language of a load/store, register-register, three-address format machine. We will use it to illustrate examples.

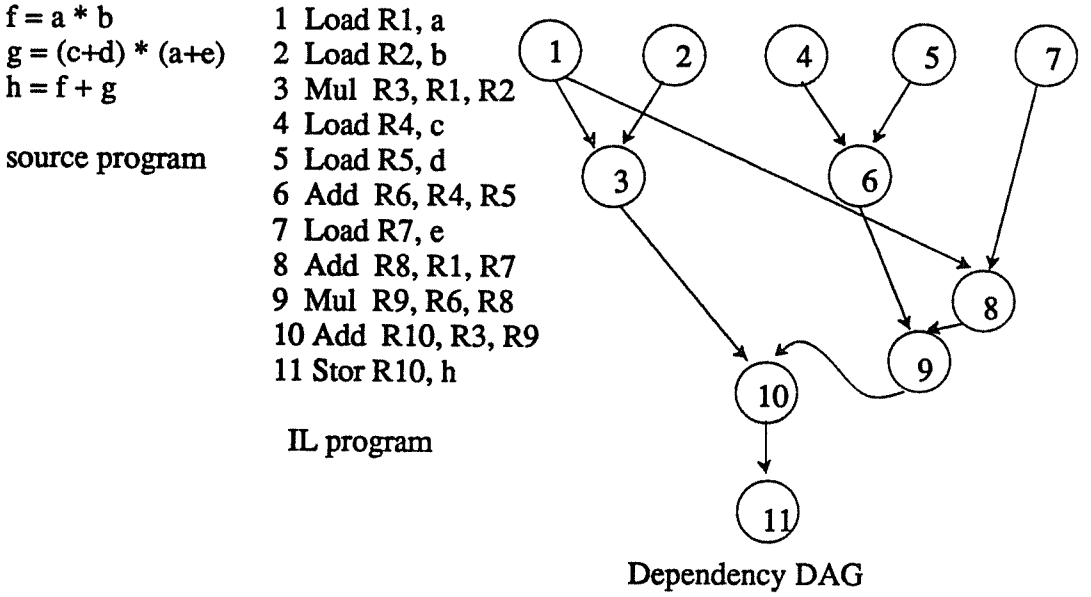


Figure 5.1 Example Program and its Dependency DAG

5.2.3. Prepass or Postpass?

Since a source language program is first translated into an IL program, code scheduling can be done either before register allocation (prepass) or after register allocation (postpass). The advantage of prepass scheduling is that the full parallelism of the program could be used. Its drawback is the overuse of registers which causes excessive register spilling. The increased instructions for register spilling will slow down the computation. Postpass scheduling does not increase spilled code, since register allocation has already been done. However, the register allocator is likely to assign the same register for unrelated instructions; sometimes, this sort of allocation is unavoidable. The reuse of registers introduces new dependency constraints, making code scheduling more restricted. An example in Figure 5.2 illustrates the pros and cons

of the above two scheduling policies. In Figure 5.2, the same program as in Figure 5.1 is used. The IL program is at the top, PR means pseudo-registers.

In the above example, we assume a stack is used to manage the register pool: dead registers are returned to the top of the stack and new registers are allocated from the top of the stack[1]. The DAG in Figure 5.1 is used for prepass code scheduling. The original DAG, which is based on the dependencies of pseudo-registers, preserves maximal parallelism. After the register allocation, the reuse of registers forces new dependencies. For example, the reuse of register 4 in instruction 7 adds a write-after-read (WAR) dependency [Rama77] from instruction 6 to 7 (Figure 5.3). This newly introduced dependency prevents instruction 7 from overlapping instruction 4 or 5, in postpass scheduling, introducing artificial pipeline delays. However, the code sequence of prepass code scheduling consumes five registers while the code sequence of postpass scheduling requires only four. If only four registers are available, the prepass code needs load and store instructions to spill registers.

5.2.4. Two Conflict Scheduling Techniques

Two code rearranging techniques could be applied during the optimization phases. They are: (1) code scheduling [Henn83, Gibb86, Youn85] to avoid delays in pipelined machines as we have discussed before — we call this technique CSP (Code Scheduling for Pipelined processors) for short, and (2) code reorganization [Davi86] to minimize the number of registers required — we call it CSR (Code Scheduling to minimize Registers usage) for short. CSP could be applied before and/or after the register

[1] Alternate allocation policies are discussed in section 5.4.

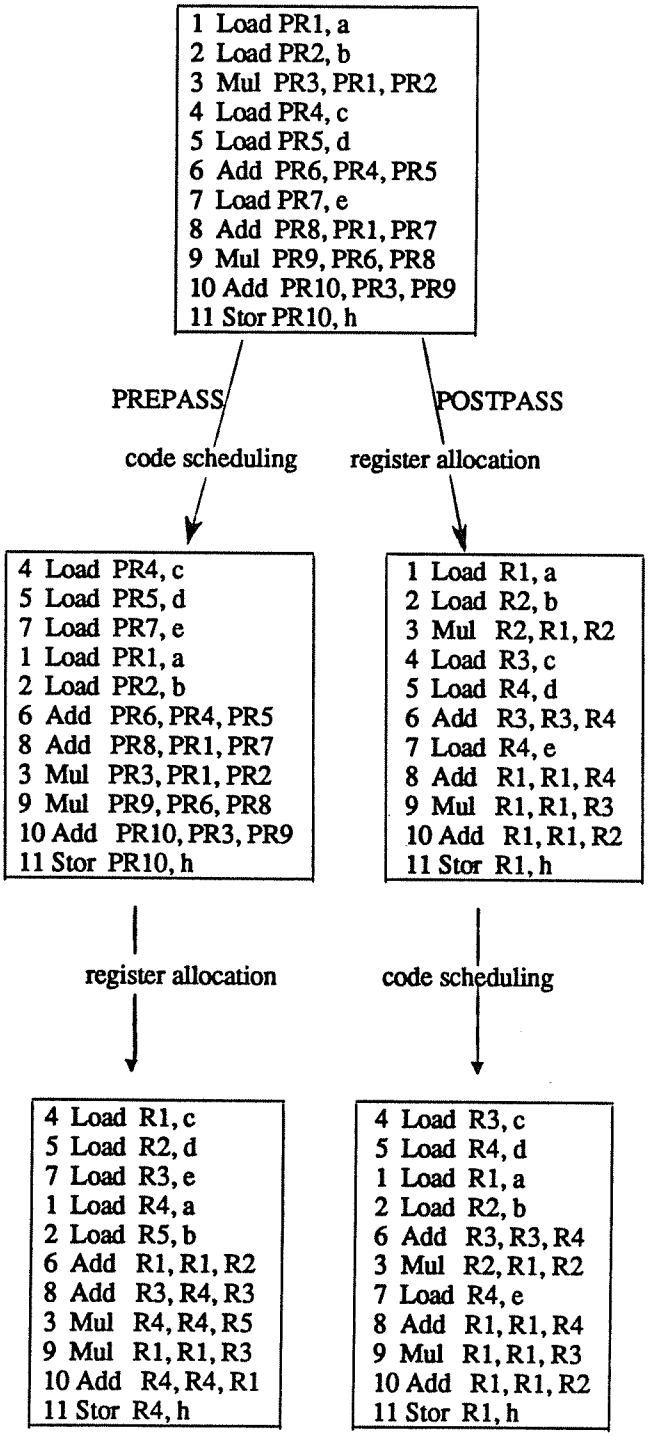
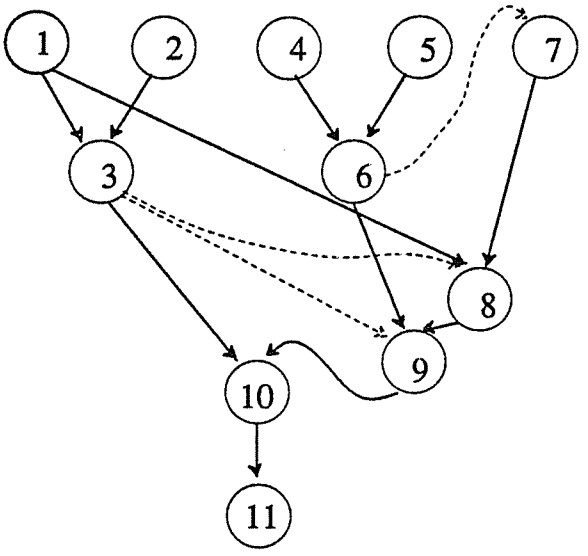


Figure 5.2 Prepass and Postpass Scheduling



solid lines -- original dependencies.

dashed lines -- dependencies added by register allocation

Figure 5.3 New Dependency Edges Added by Register Allocation

allocation phase while it only makes sense to apply CSR before register allocation. The use of CSP has been discussed; we now introduce the use of CSR in the following example.

As in Figure 5.2, a typical register allocation for the IL program requires four registers. The rearranged code sequence using CSR (as in Figure 5.4) needs only three registers.

The rearranging technique that generates the above code sequence has been used in the code generation phase to determine better evaluation orders of expression trees or DAGs. The well-known Sethi-Ullman algorithm [Seth70] generates the optimal evaluation order (using minimal number of registers) of expression trees. Heuristic algorithms are also available for DAGs with common subexpressions [Aho77].

```
4 Load PR4, c                                    4 Load R1, c
5 Load PR5, d                                    5 Load R2, d
6 Add  PR6, PR4, PR5                             6 Add  R1, R1, R2
1 Load PR1, a                                    1 Load R2, a
7 Load PR7, e                    after           7 Load R3, e
8 Add  PR8, PR1, PR7   == register allocation ==> 8 Add R3, R2, R3
9 Mul  PR9, PR8, PR6                             9 Mul  R1, R3, R1
2 Load PR2, b                                    2 Load R3, b
3 Mul  PR3, PR1, PR2                             3 Mul R3, R2, R3
10 Add  PR10, PR3, PR9                           10 Add  R1, R3, R1
11 Stor PR10, h                                  11 Stor R1, h
```

Figure 5.4 CSR Minimizing the Use of Registers

Recently, Davidson [Davi86] has separated this optimization technique from the code generation phase and implemented it as an independent code reorganization technique. The key idea of this optimization is to prevent a register from holding a temporary too long. Hence the number of simultaneously live registers could be reduced.

CSP and CSR conflict with each other. CSP tends to increase the lifetime of each pseudo-register while CSR wants to shorten it. If CSP and CSR are implemented in different optimization phases, they will interfere with each other. Using one or the other technique on a per block basis may yield poor results when the number of available registers varies within a basic block. We propose, therefore, to integrate CSP and CSR into a single phase so that they will be friends instead of foes.

5.3. A Solution for Prepass Scheduling

The major disadvantage of prepass scheduling is that it may overuse registers causing register spilling. We propose to integrate CSP and CSR in prepass code scheduling to control register spilling. The basic idea is to keep track of the number of

available registers during code scheduling. Since each issued instruction may create a new live register and terminate the lifetime of some registers, we can count the number of available registers. When there are enough registers, the scheduler uses CSP to reduce pipeline delays. When the number of available register is getting low, the scheduler switches to CSR to control the use of registers. The following example explains this approach.

Example

Suppose the input program is the same as in Figure 5.1 and there are four[2] registers available for this program. Our new code scheduler will schedule the program in a sequence like:

4 Load PR4, c
5 Load PR5, d
7 Load PR7, e
1 Load PR1, a

The scheduler must now choose between issuing instruction 2, which activates one register, and instruction 6, which frees one net register. Since the available registers have been used up, CSR takes charge of scheduling, and issues instruction 6. We then return to CSP and instruction 2 is issued after instruction 6. The complete reorganized code sequence is as in Figure 5.5.

Notice that this code sequence uses four registers, the same number as the postpass code sequence used. Compared to the postpass code sequence as in Figure 5.2,

[2] A machine typically has eight or 16 general purpose registers. However, we assume that other registers have already been preallocated to frequently used variables or constants and thus only four are left for this basic block.

```
4 Load PR4, c                                    4 Load R1, c
5 Load PR5, d                                    5 Load R2, d
7 Load PR7, e                                    7 Load R3, e
1 Load PR1, a                                    1 Load R4, a
6 Add  PR6, PR4, PR5                             6 Add  R1, R1, R2
2 Load PR2, b          == register allocation ==> 2 Load R2, b
8 Add  PR8, PR1, PR7                             8 Add  R3, R4, R3
3 Mul  PR3, PR1, PR2                             3 Mul  R4, R4, R2
9 Mul  PR9, PR6, PR8                             9 Mul  R1, R1, R3
10 Add PR10, PR3, PR9                            10 Add R4, R4, R1
11 Stor PR10, h                                  11 Stor R4, h
```

Figure 5.5 Code Sequence Using Integrated Scheduling

however, this code sequence has fewer runtime interlocks.

5.4. Implementation Notes

5.4.1. CSP, CSR and AVLREG

There are two major parts in our approach: CSP and CSR. The use of CSP in our work is based on the work by Young [Youn85]. We also take ideas from others [Gibb86, Henn83] to improve the CSP algorithm. Young [Youn85] assumes the target machine has multiple functional units whose pipelines vary in length. Instructions complete whenever they leave their particular functional unit pipelines. The estimated execution time of each instruction is used to compute the cumulative cost of each node in the DAG. This cumulative cost identifies which node is on the critical path during instruction scheduling. Instructions are scheduled in a topological sort order of the DAG. Nodes on the current critical path have higher issue priority. In contrast to [Henn83], hardware interlocks are assumed rather than using software to enforce

interlocks. We also generalize the way of checking interlocks at code scheduling time from Gibbons [Gibb86] and Hennessy [Henn83] in our scheme.

The CSR used in our approach is quite different from earlier work [Seth70, Aho86], which determines the complete evaluation order of an expression tree [Seth70] or a DAG (having common subexpressions) [Aho86] to minimize the number of registers used. In our approach, when CSR is called, the evaluation order of the DAG has been partially determined (some nodes have been issued). The goal of CSR at this point is to find the next instruction which will not increase the number of live registers, or if possible, decrease that number. Our CSR does not decide the total evaluation order. The basic approach of our CSR is to find an instruction that frees more registers than the number of live registers it creates. When no such instructions exist, the scheduler looks for instructions on partially evaluated paths, since once the partially evaluated path is fully evaluated, registers may be freed.

Switching between CSP and CSR is driven by the number of available register, AVLREG. CSP is responsible for code scheduling most of the time. When AVLREG drops below a threshold (say, one) CSR is invoked. After AVLREG is restored to an acceptable value, CSP resumes scheduling. AVLREG is initially determined by the total number of registers minus the number of registers live-on-entry. We assume a global data flow analysis [Aho86] supplied the information of registers live-on-entry. Reference counting is used to determine when pseudo-registers are dead and can be freed. We increase AVLREG when there are freed registers, and decrease AVLREG when instructions create live registers.

5.4.2. Renaming of Pseudo-Registers

A single assignment rule— every pseudo-register is written only once statically — is used to maintain maximal scheduling flexibility. An intuitive implementation is to assign a new pseudo-register whenever there is a write. This implementation has problems with local variables. For example, in Figure 5.6, if X is allocated to different pseudo-registers in disjoint blocks, which pseudo-register should be referenced by X in the join block? To avoid this problem, we assign a unique pseudo-register for each local variable (if the variable has no possible aliases), and a unique pseudo-register for every newly created temporary. Since a local variable may be written more than once in a basic block, a renaming procedure is performed to enforce single assignment. This renaming makes reference counting realizable. The example in Figure 5.7 shows the renaming of local variables.

Renaming simplifies detection of dependencies. Since each pseudo-register is only written once, there should be no WAR (write-after-read) and WAW (write-after-write) dependencies at the pseudo-register level. However, WAR detection is still needed for the case depicted in Figure 5.8, where pseudo-register X is live-on-entry to the basic block and will be written later in the basic block.

5.4.3. Interlock Checking at Scheduling Time

In load/store, register-register architectures, all interlocks can be resolved at the instruction issue stage [Kogg81, Cray82]. The code scheduler can use instruction cycle time estimates to minimize execution delays due to interlocks resulting from varying

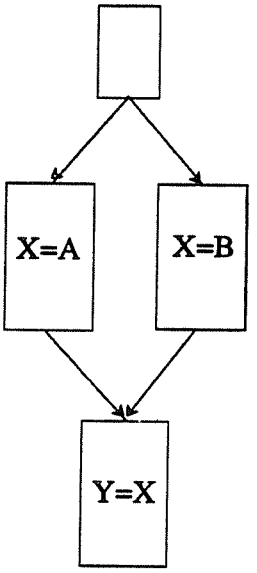


Figure 5.6 Local Variables Make Single Assignments Non-Trivial

```
X = A                  Move PR1, PR2                    Move PR11, PR2
B = X + C              Add  PR3, PR1, PR4               Add  PR13, PR11, PR4
X = D + B    =>        Add  PR1, PR5, PR3   == renaming => Add  PR1, PR5, PR13
B = X + E              Add  PR3, PR1, PR6               Add  PR3, PR1, PR6
```

Figure 5.7 Renaming of Local Variables

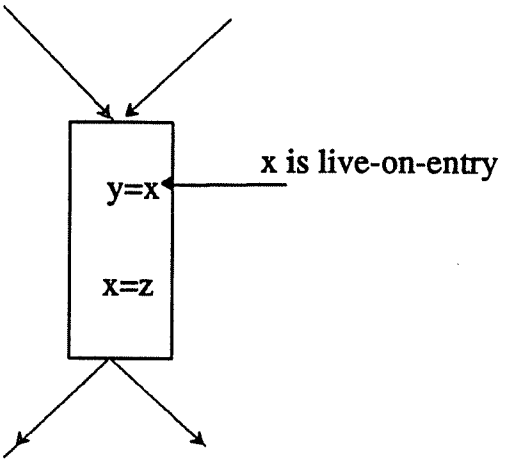


Figure 5.8 WAR Detection for Live-On-Entry Variables

length functional unit pipelines. At code scheduling time, each issued instruction reserves a destination register (primary register receiving the computation result, if any). If this instruction takes N cycles to execute, then the number N is put in the reservation table. After each instruction is issued, all positive numbers in the reservation table are decreased by one. We say an instruction "will have interlock" as long as any of its primary registers are reserved. The maximum "cycles remaining" value of the primary registers is the length of the instruction's interlock. Since we assume hardware interlock resolution, estimates do not have to reflect hardware behavior exactly.

5.4.4. Leader Set and Ready Set

A leader of a DAG is a vertex with no predecessors. An instruction may not be issued until it becomes a leader. As instructions are issued, their nodes are removed

from the DAG and some successor nodes become new leaders. All leaders are maintained in a *leader set*. Instructions in the leader set lacking interlocks with previously issued instructions are promoted from the leader set to a *ready set*. All the instructions in the ready set are ready to be issued.

5.4.5. Integrated Scheduling Algorithm

(1) Rename pseudo-registers to enforce single assignment.

(2) Read in the basic block, create the DAG and calculate the reference count of each pseudo-register.

(3) Compute the cumulative cost of each node in reverse topological sort order.

(4) Issue instructions in topological sort order.

Here are details of step 4:

```
4.0 Calculate the leader set.

while (leader set or ready set is not empty) do

  4.1 Move nodes with no interlocks from the leader set to the ready set.
  4.2 if (AVLREG > threshold value) then
          if (ready set is not empty) then
            select one node from the ready set with maximum cumulative cost.
          else
            select one node from the leader set with maximum cumulative cost.
          endif
      else {invoke CSR}
          if there are nodes in the ready set that can free registers then
            select one node which frees the most registers.
            if there are more than one such node then
                  select one with maximum cumulative cost.
            endif
          else
            if there are nodes in the leader set that can free registers then
                  select one which frees the most registers.
```

```
                if there are more than one such node then
                   select one that has the fewest interlocks.
                endif
            else
                find a partially evaluated path, (for example, one of its
                RAW dependency has been lifted)
                select one node from the leaders of this path.
                if there are no such partially evaluated paths then
                   select any one node from the ready set or leader set (if the
                   ready set is empty).
                endif
            endif
        endif
    endif

  4.3 Issue the selected instruction
    if the issued instruction creates one live register then
        decrement AVLREG by 1.
    for each pseudo-register referenced in this instruction do
        decrement its reference count by 1
        if the reference count drops to 0 then
           increment AVLREG by 1.
        endif
    end for
    Remove this instruction from the DAG
    Remove all dependencies caused by this instruction
    Reserve the destination register in the reservation table.

  4.4 Insert new leaders into the leader set

end while
```

The input program used in this example is the same one as used in section 2. In addition, we assume the following timing for the relevant functions.

Function	Timing(clock periods)
Load	4
Store	1
Add	2
Multiply	3

It may take more than one clock period to finish a store operation. However, as far as the issue logic is concerned, a store does not cause any subsequent instructions to be blocked from issuing. We also assume that the initial value of AVLREG is 4. The weighted DAG is shown in the Figure 5.9 The code sequences generated by prepass code scheduling, postpass code scheduling, and our algorithm are shown in Figure 5.10. Notice that, since the number of available registers is 4, prepass code scheduling incurs spilling costs.

5.4.6. A Variation on Profitable Register Spilling

In the above algorithm, we have assumed that register spilling is more costly than run-time interlocks. This assumption may not be true for highly pipelined machines.

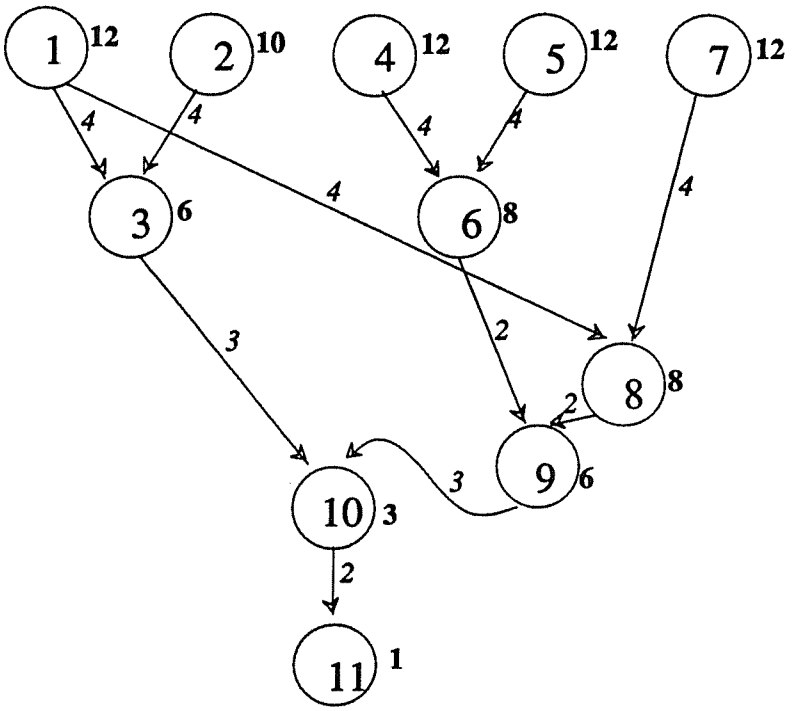
(1) In highly pipelined machines, a pipeline interlock could be very long compared to the issuance of a couple of spill instructions.

(2) Spill code can often be scheduled to run in otherwise wasted cycles.

(3) Registers that have not been changed (e.g. registers containing loop invariants), can be spilled at a lower cost.

(4) The inserted load/store instructions can be moved to reduce the urgency of data dependency. In the PL.8 compiler this is done by a second pass code scheduling (additional code scheduling done after register allocation).

Since some machines make spilling relatively less expensive, (*e.g.* in the CRAY-1, spilling A register to B register is not costly) we have developed a variation of the algorithm to consider profitable register spilling. The variation operates as follows: when available registers are running out and the next selected instruction has a long interlock with a previously issued instruction, the scheduler checks if there is a live



Weighted DAG

boldface numbers associated with nodes are cumulative costs
italic numbers associated with edges are execution time estimates

Figure 5.9 Weighted DAG Identifying Critical Paths

Prepass	Postpass	Integrated
1 Load R1, a	4 Load R3, c	1 Load R1, a
4 Load R2, c	5 Load R4, d	4 Load R2, c
5 Load R3, d	1 Load R1, a	5 Load R3, d
7 Load R4, e	2 Load R2, b	7 Load R4, e
s Stor R4, stacktop+1	6 Add R3, R3, R4	6 Add R2, R2, R3
2 Load R4, b	3 Mul R2, R1, R2	2 Load R3, b
6 Add R2, R2, R3	7 Load R4, e	8 Add R4, R1, R4
s Load R3, stacktop+1	8 Add R1, R1, R4	3 Mul R1, R1, R3
8 Add R3, R1, R3	9 Mul R1, R1, R3	9 Mul R2, R2, R4
3 Mul R1, R1, R4	10 Add R1, R1, R2	10 Add R1, R1, R2
9 Mul R2, R2, R3	11 Stor R1, h	11 Stor R1, h
10 Add R1, R1, R2		
11 Stor R1, h		
(22 cycles)	(20 cycles)	(18 cycles)

Figure 5.10 Comparisons of Scheduled Code Sequences

pseudo-register which could be spilled at a low cost. If there is at least one such register, the scheduler will revert to CSP scheduling to favor instructions having no interlocks and/or are on the critical evaluation path. Later in the register allocation phase, the register allocator will take care of the spilling job.

Because of the uncertainties involved in predicting how inserted load instructions will interfere with subsequent register usage, we have only attempted to guess at a good threshold value for determining when spilling is worthwhile.

5.5. Scheduling Loads and Stores

In load/store architectures, only load and store instructions are allowed to access memory. They need special treatments since they are accessing resources other than the register file. We can simply assume the memory is a single resource. Under this

assumption, each load instruction is logically dependent on all previous store instructions, and each store instruction is dependent on all previous loads and stores. This simple assumption introduces minimal complexity but maximal restriction to the code scheduling. Relaxing this assumption will certainly improve the performance of code scheduling but the problem of aliases is difficult: how to distinguish two memory references at compile time. Unless there is enough information to distinguish memory references, (for example, using an anti-alias analyzer) the compiler can only make the worst case assumption that every memory reference is an alias of others.

Gibbons [Gibb86] makes an assumption that load/stores using different base address registers are referencing different memory locations in his code scheduler. In our experiment, we assume there is information to distinguish memory objects. For some memory objects that may be aliased with others, preserved pseudo-registers are used to compute their addresses. Load and store instructions using the preserved pseudo-registers are assumed to be dependent and will be scheduled strictly in their logical order.

So far we have assumed an interleaved memory system so that several memory access operations can be overlapped. Memory access conflicts and access hazards often degrade the performance of the memory system. A memory access conflict occurs when a memory request is accessing a busy module. A memory access hazard exists when more than one memory request references the same memory location, and an incorrect sequence of memory operations can result in using wrong data or storing wrong information into memory. Due to insufficient information, compilers can only enforce load and store instructions in their logical order but not schedule them to avoid access hazards. However, a well-designed memory controller can schedule memory access

operations at run-time to reduce access conflicts and resolve access hazards [Liou85].

5.6. Simulation Studies and Discussion

5.6.1. Simulations

In this section, we show some experimental results concerning the effectiveness of our new approach. An interpreter and a simple performance simulator have been built to evaluate how fast instructions can be issued for a hypothetical machine. The hypothetical machine is the same as that used in examples of the previous sections. Its architecture has a load/store, register-oriented, three-address instruction format. It has a single general purpose register file. The number of registers and the degree of pipelining of the machine can be varied by changing the parameters in a profile. The pipelined implementation of our model architecture is shown in the Figure 5.11. The hypothetical machine has hardware hazard detection and an interlock mechanism. We assume in the machine that data dependencies are the only reason to block instructions from issuing. There are, of courses, other reasons to stop the smooth flow of a pipeline. However, they are beyond the scope of this chapter.

We use the first twelve Livermore loops [McMa72] for benchmark programs. Loop unrolling techniques [Dong79] have been used to obtain large basic blocks. Since unrolled loops may overflow an instruction buffer, decreasing performance, we unroll loops until their program size is a little less than some predefined limit. In our simulation, this limit is 32 instructions. Although 32 is relatively small compared to the instruction buffer in modern supercomputers (CRAY-1 has a buffer size of 256, CRAY-XMP has a size of 512), it is large enough to study the interdependency between

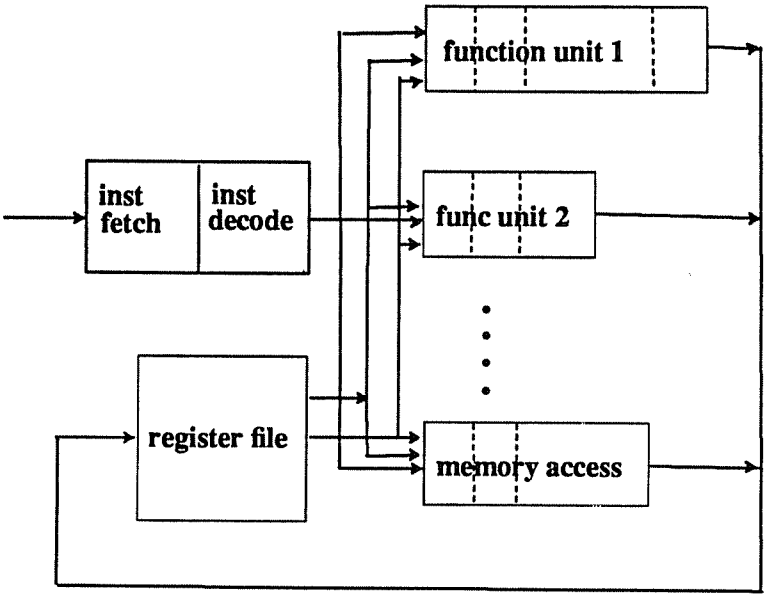


Figure 5.11 Pipelined Implementation of Our Model Architecture

code scheduling and register allocation. All of the loops are translated into the IL of the hypothetical machine with standard optimizations [Ausl82] done by hand.

Different approaches for code scheduling that have been tested in the simulation are:

(1) Prepass: IL --> CS --> RA

(2) Twopass: IL --> CS --> RA --> CS

code scheduling is performed both before and after register allocation. The second pass scheduling is primarily for the inserted load/stores introduced by register spilling.

(3) Postpass: IL --> RA --> CS

(4) Postpass with round-robin register allocation (PostRR for short):

The regular register allocation uses a stack to manage register reuse. Some papers [Henn83, Youn85] suggest using round-robin allocation, which cycling through registers, to reduce the inadvertent dependencies introduced by register reuses. Hence we implemented PostRR as a variation of general Postpass. The code scheduling algorithm for approaches (1) through (4) is essentially the CSP part as discussed in section 5.4.

(5) Ours0: As described in the section 5.4.6.

This algorithm does prepass code scheduling with appropriate control of register usages.

(6) Ours1: A variation of Ours0.

This version considers spilling a register as an alternative when the next issuing instruction has long interlocks with previously issued instructions.

All the above approaches used the same register allocator, which uses a replacement-based algorithm as we have discussed in chapter 3.

In Figure 5.12, we present the relative performance of six different approaches. The performance measure is the number of clock cycles needed to issue and execute a program. The number of available registers is varied from 4 to 30. The machine is assumed to be heavily pipelined (HP for short), similar to the CRAY-1 [Cray82]. We assume 11 clock periods (CP) for a load, 3 CPs for an add , 6 CPs for a multiply, and so on. Figure 5.13 is similar to Figure 5.12 except that the machine is not so heavily

pipelined. In Figure 5.13, a load takes 6 CPs, an add takes 2 CPs, an multiply takes 3 CPs, and so on.

Figures 5.12 and 5.14 share machine assumptions, as do Figures 5.13 and 5.15. In Figures 5.14 and 5.15, the measure is the number of instructions. Since prepass code scheduling often results in register spilling, the sizes of its resulting programs are usually larger than programs of postpass scheduling and our scheduling scheme.

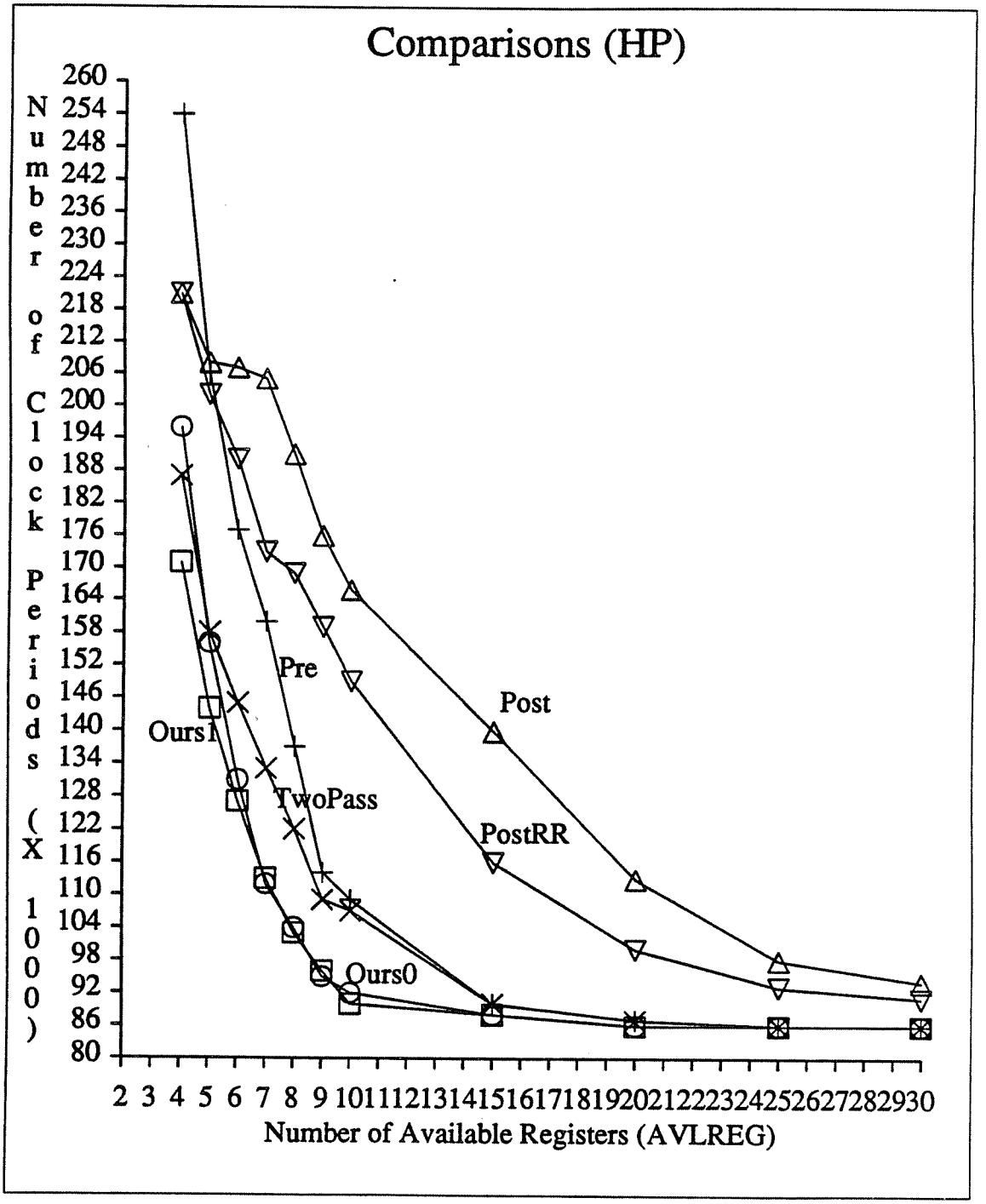


Figure 5.12 Comparisons of Execution Cycles (Highly Pipelined Model)

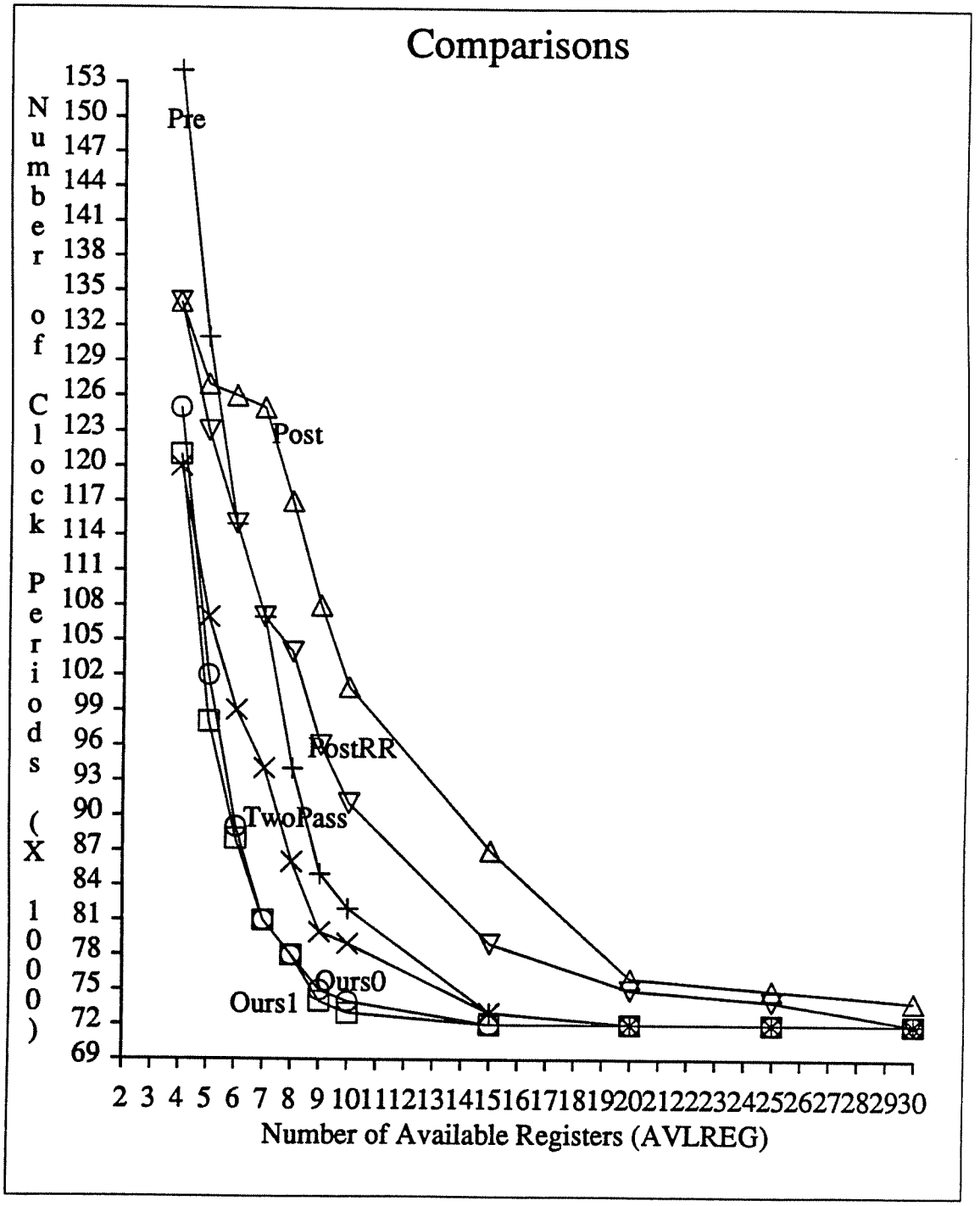


Figure 5.13 Comparisons of Execution Cycles (Medium Pipelined Model)

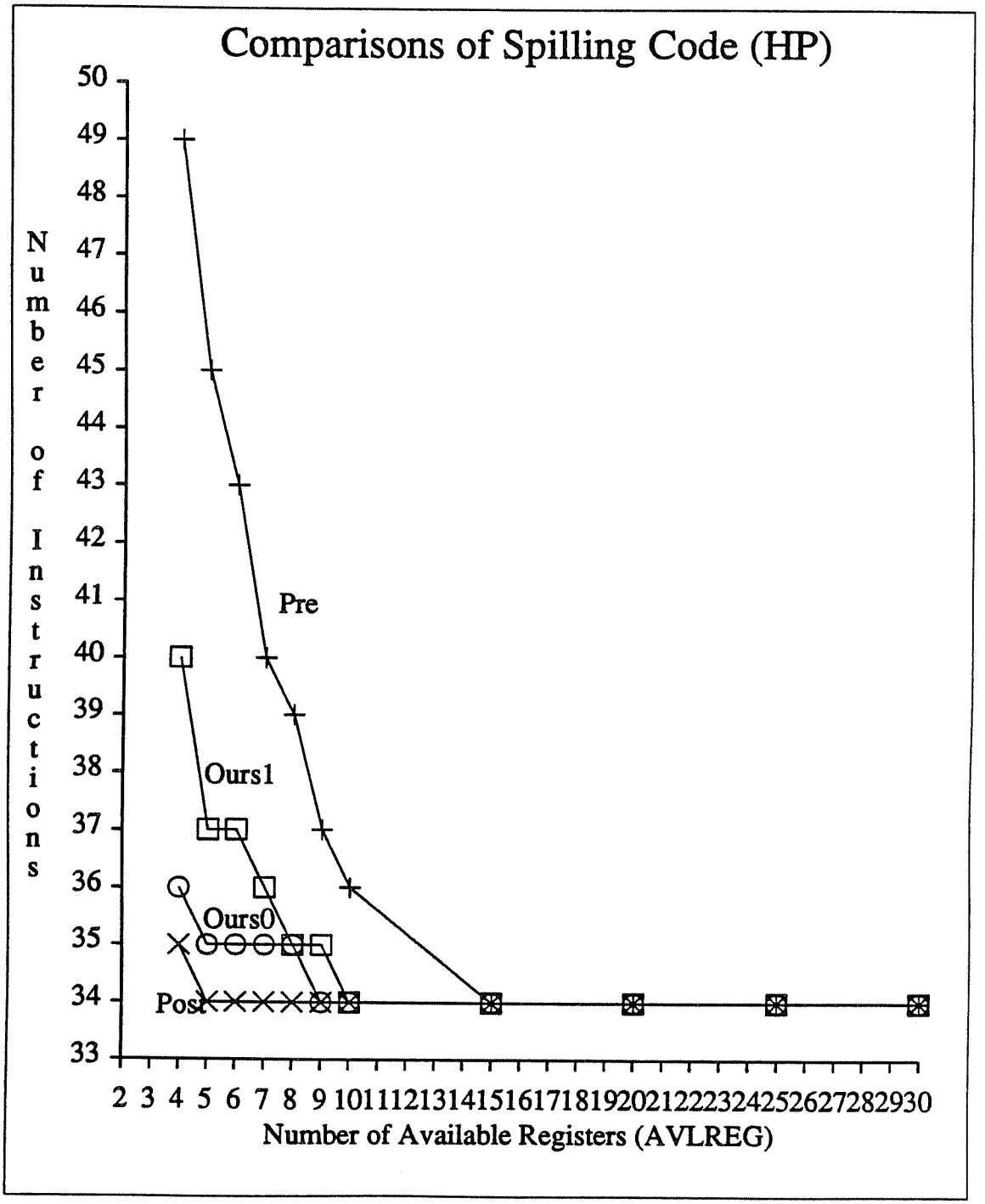


Figure 5.14 Comparisons of Spill Code (Highly Pipelined Model)

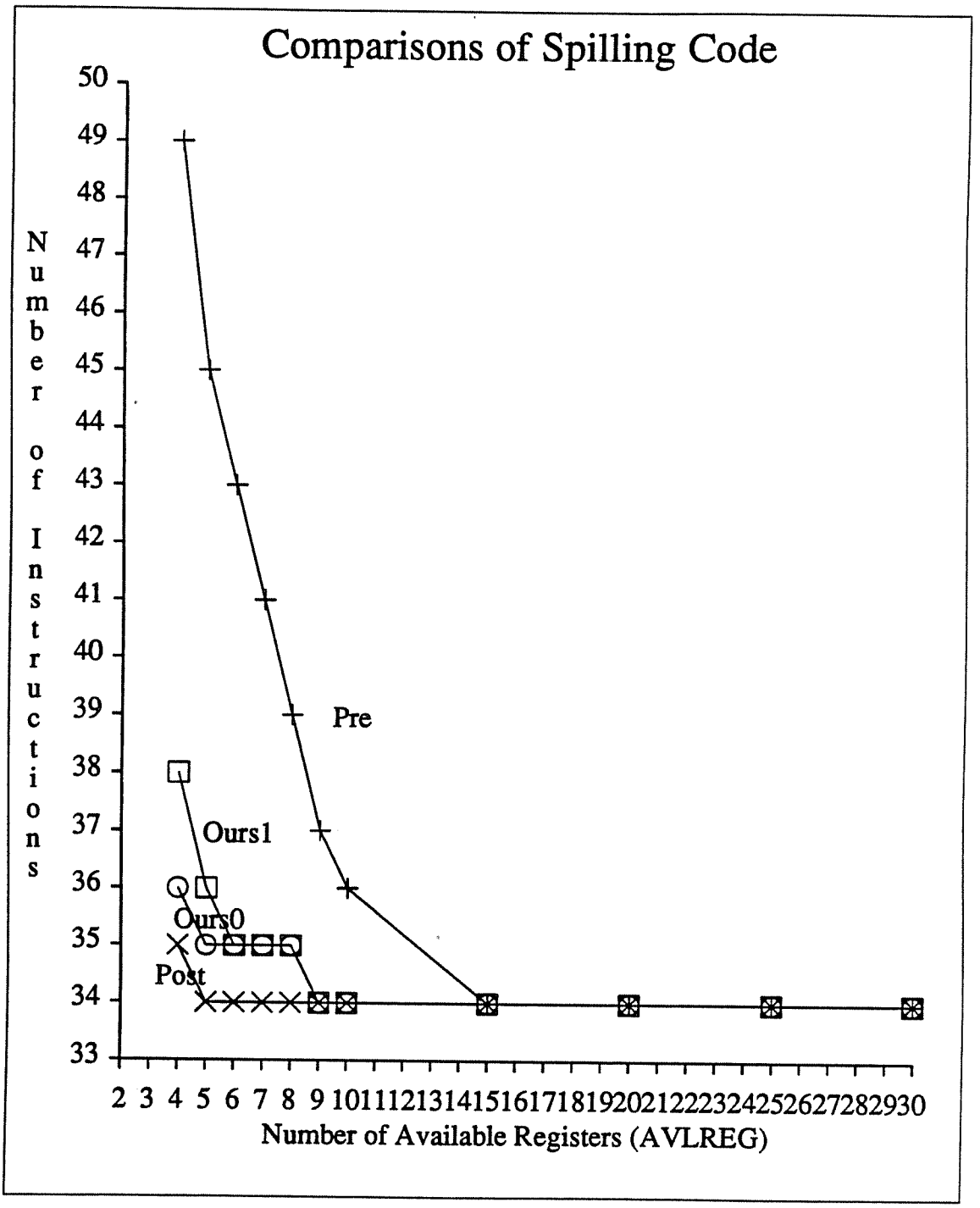


Figure 5.15 Comparisons of Spill Code (Medium Pipelined Model)

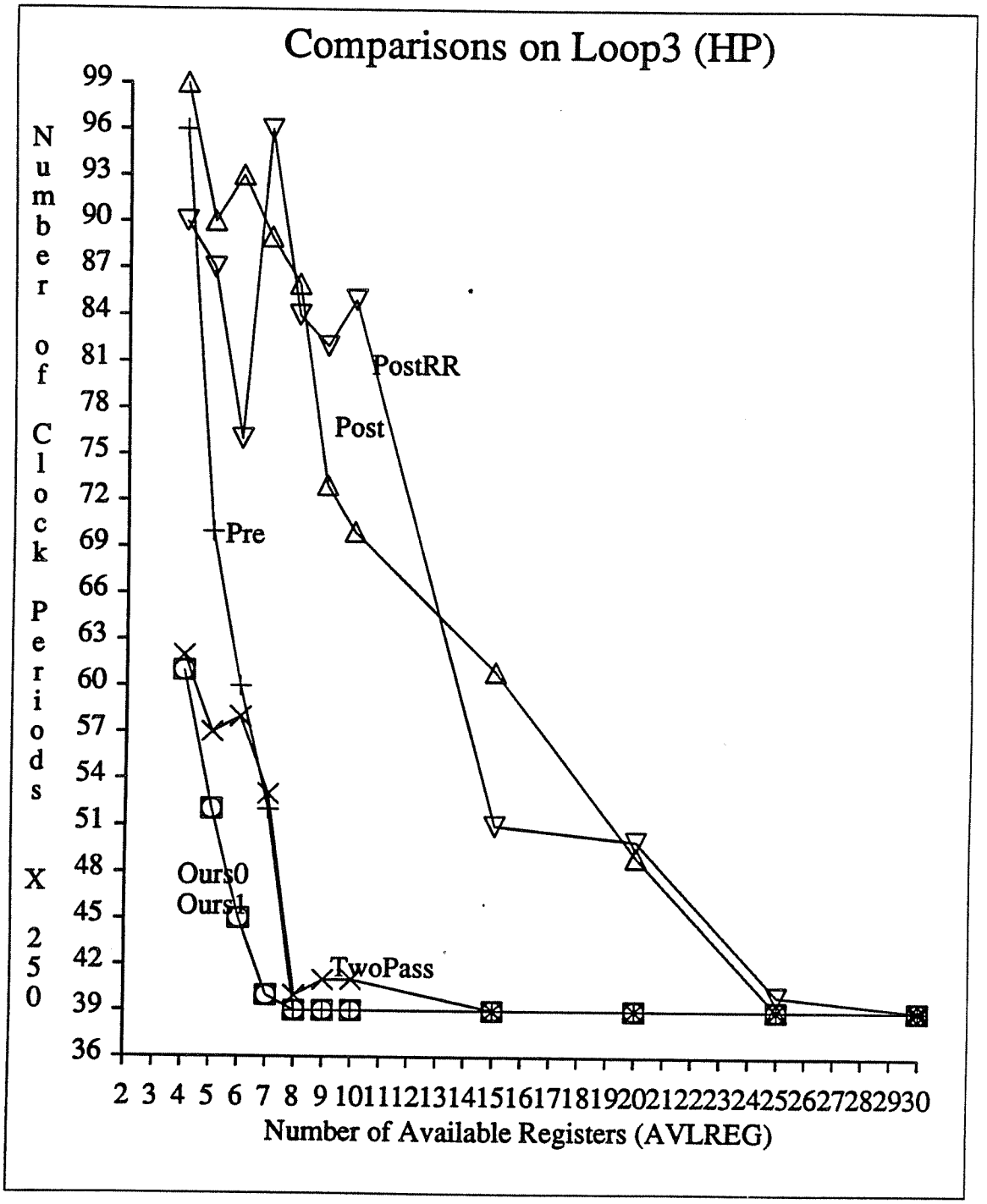


Figure 5.16 Comparisons Based on A Single Loop

5.6.2. Discussion

5.6.2.1. General Discussion

Using Figures 5.12 through 5.15, we make the following observations.

(1) In Figures 5.12 and 5.13, prepass code scheduling usually has better performance than postpass scheduling unless the number of available registers is very low. This is because prepass scheduling has much better flexibility to schedule code, especially when more parallelism exists. However, we should not conclude that prepass scheduling is better than postpass scheduling. In Figures 5.14 and 5.15, the prepass scheduled programs have significantly larger size than the programs of postpass scheduling. Larger programs may execute more slowly, since larger loops are more likely to overflow the instruction buffer, and inserted load/store instructions increase the number of memory fetches and stores. Additional load/stores will slow the computation seriously if memory bandwidth is the performance bottleneck.

Our approach Ours0 outperforms both prepass scheduling and postpass scheduling as shown in Figure 5.12 and 5.13 while the additional load/store instructions of Ours0 are only slightly (less than 3%) more than postpass scheduling, as shown in Figure 5.14 and 5.15. Therefore, we believe our approach has better performance, in general than the others.

(2) In theory, if there are an infinite number of registers, all the different approaches have the same performance. As shown in in Figures 5.12 and 5.13, all the curves level off to the same point when the number of available registers becomes large.

However, minimizing the number of registers is critical for designing high-performance computers; for register access to be fast, the size of the register file should be small [Henn84]. Hence, the more important question is how to use registers efficiently, so that we do not need a very large register file. As shown in Figures 5.12 and 5.13, the curve of our approach levels off quickly, implying that better scheduling can use a (relatively) small number of registers efficiently.

A register file can be used to hold temporaries and frequently used variables. An efficient algorithm uses fewer registers for temporaries, leaving more registers for frequently used constants and variables. The number of registers is well-defined in an architecture. Yet the architecture can have quite different implementations. A highly pipelined implementation requires more registers for temporaries so that interlocks can be reduced. When a highly pipelined implementation is used, the approach of using better algorithms to make effective use of the register file is more favorable than the approach of redesigning the architecture with more registers.

Advances in silicon technologies may make it relatively easy to have a large number of registers on-chip in the near future. However, for GaAs technology, which is much faster than silicon technology, the amount of on-chip memory allowed is very limited [Milu86]. In a technology hierarchy or a memory hierarchy, the space in the top level is always limited, and efficient algorithms are necessary to make effective use of the scarce space.

(3) The variation of our algorithm, Ours1, which considers register spilling as an alternative when the issuing instruction has long interlocks with previously issued

instructions, has slightly better performance than Ours0 in Figure 5.13. As in Figure 5.12, Ours1 has significantly better performance only when the number of registers is very low and when the machine is highly pipelined. Although Ours1 does not have an impressive performance improvement over Ours0 in general, we still believe register spilling is an important alternative when the machine is heavily pipelined. Live-through variables, which are live on entry and live on exit in a basic block but are not referenced in the block, are good candidates for spilling when there is a need for free registers. Loop invariants may also be spilled at a low cost.

An inserted load instruction may have a long interlock with the next instruction which uses the operand. Therefore, it is important to reschedule the inserted load and store instructions. Since the algorithm Ours1 does not use a second pass to reschedule code, it might be expected to suffer from a performance degradation caused by those inserted load and stores. This is not the case because the register allocator will do partial scheduling at the time the load/stores are inserted. This partial scheduling is performed as follows: when a load (or a store) instruction is inserted, the register allocator looks at the generated code sequence backwards to find a place which is legal yet away from the current use. This simple placement method is much cheaper than a complete rescheduling process.

(4) Some compilers designed for pipelined processors use round-robin register allocation, which cycles through the registers available for use. Intuitively, this allocation policy seems to avoid the situation of having a long evaluation path due to the intensive reuse of certain registers. In Figure 5.13, postpass scheduling with round-robin allocation does outperform stack allocation most of the time. It is

also true in Figure 5.12 except when the number of available register is low. Since every reuse of a register will add some WAR dependencies to the DAG, combining two parallel evaluation paths into a sequential one, without having the detailed information of the DAG, no allocation policy will be uniformly superior to others in balancing the length of merged evaluation paths. One alternative to the current approach is to provide DAG information to the register allocator. With the DAG information, the register allocator may be able to reuse registers in a way such that the depth of the new DAG can be minimized. In the next chapter, we will introduce such an approach.

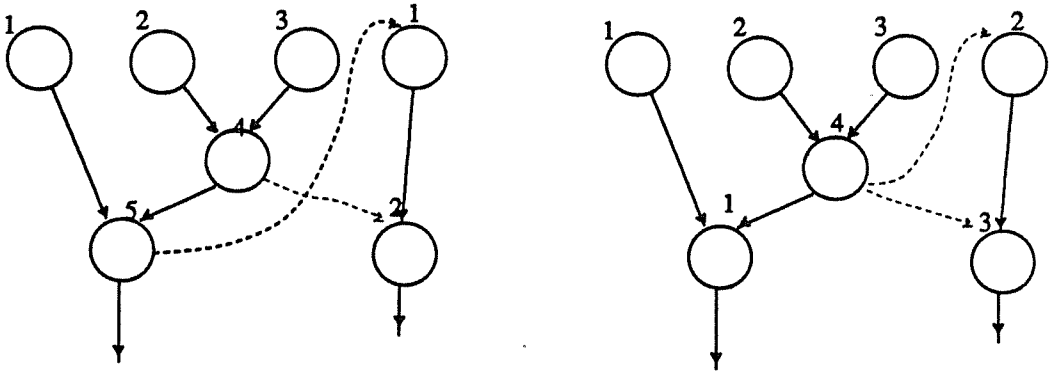
Some anomalies exist in Twopass and Postpass scheduling experiments. Since they can not be observed from Figure 5.12 and 5.13 which are figures averaged over 12 loops, Figure 5.16 is used to present the curves of a single loop. The following discussions are based on Figure 5.16.

(1) The Post curve in Figure 5.16 exhibits a deterioration in performance when the number of registers is increased from five to six. The PostRR curve exhibits a similar anomaly for the transition from six to seven and nine to ten registers. These anomalies result from variations in induced dependencies, as illustrated in Figure 5.17. The DAG on the left hand side has five available registers while the DAG on the right hand side has only four. Round-robin register allocation was used. Notice that after register allocation, the left DAG is more restricted than the right DAG. Because the right tree in the left DAG can not be interleaved with the left tree any more, instructions are forced to be executed in a sequential order.

Figure 5.16 also showed that sometimes Post (stack register allocation)

outperforms PostRR (round-robin register allocation) and vise versa. This supports our previous assertion that no allocation policy will be superior to others without detailed DAG information.

(2) Two-pass scheduling is used to improve the performance of prepass scheduling. The second scheduling pass is used to schedule the inserted load/store instructions. In Figures 5.16, the anomaly of the Twopass curve occurs when the number of registers is eight, nine and ten. Not only is the performance poorer than the single pass scheduling but the curve also exhibits a deterioration in performance when the number of registers is increased. When there are enough registers, no inserted load/stores are required for spilling. When there are no inserted load and store instructions, a second pass scheduling is useless. This confirms the correctness of



solid lines -- original dependencies.

dashed lines -- dependencies added by register allocation

Figure 5.17 One Explanation of the Anomaly in Postpass Scheduling

the PL.8 compiler in applying a second pass scheduling only when there are inserted load/stores. The performance deterioration anomaly is due to the effect of register allocation as we have explained previously.

5.6.2.2. Total *vs* Available Registers

In the simulation studies, the number of available register is used rather than the total number of registers. Since each loop needs a different number of registers to be allocated for the loop invariants (e.g. base addresses, constants) and the frequently used variables (e.g. loop index variables), if the total number of registers is used, the number of available registers for each loop will be different. In order to observe a more consistent behaviour, all loops are assigned the same number of available registers to start with. For example, loop two needs four registers allocated across the loop and loop eight needs 20; if eight registers are assigned to loop one, then 24 registers should be assigned to loop eight since they can both having four available registers to start with. In order to study the interplay between code scheduling and register allocation, using the number of available registers makes more sense. However, it is informative to see how the loops work with a total number of registers, because in real machine simulations, the machine resources (registers) will not be changed with different application programs.

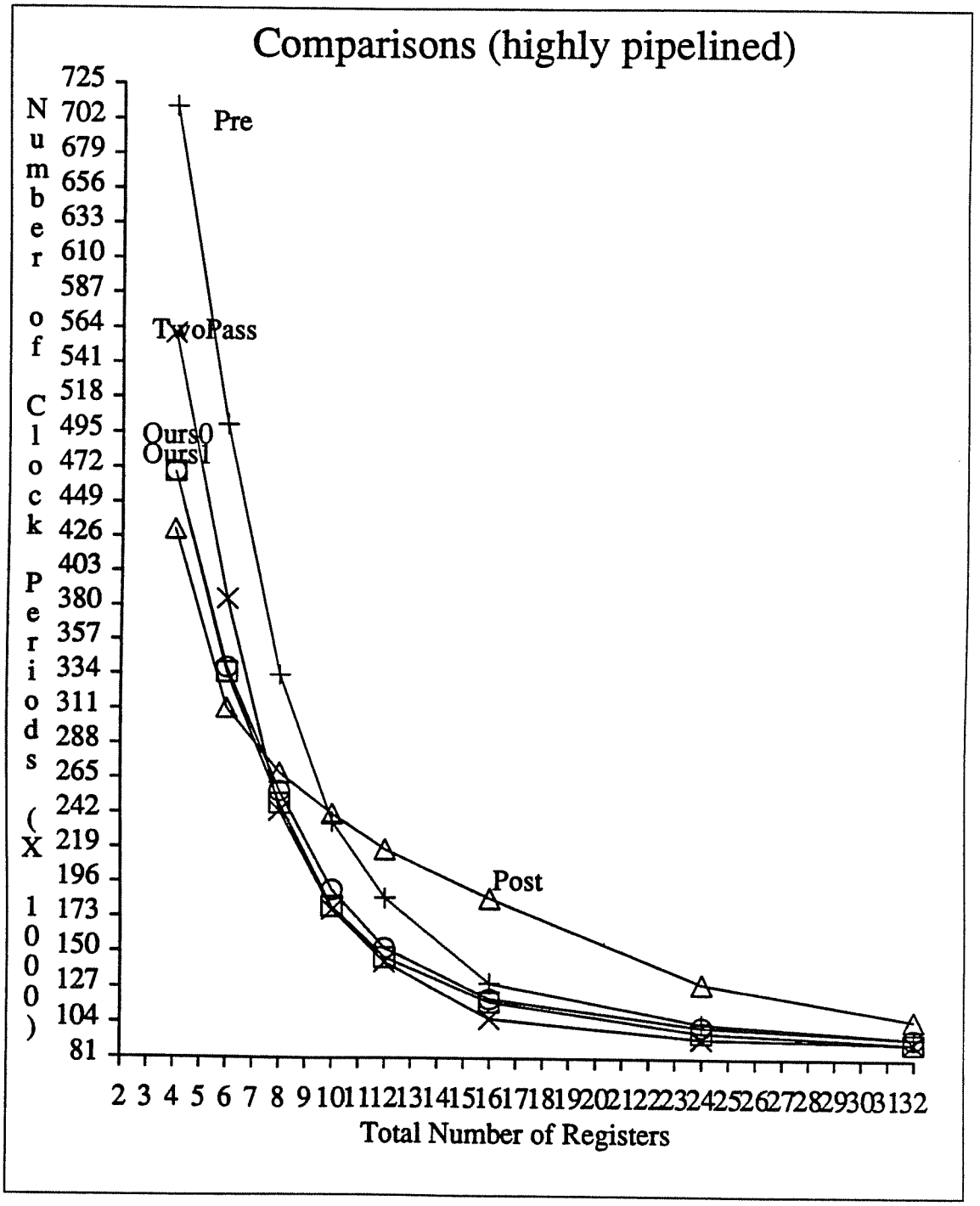


Figure 5.18 Comparisons Using Total Number of Registers (Highly Pipelined)

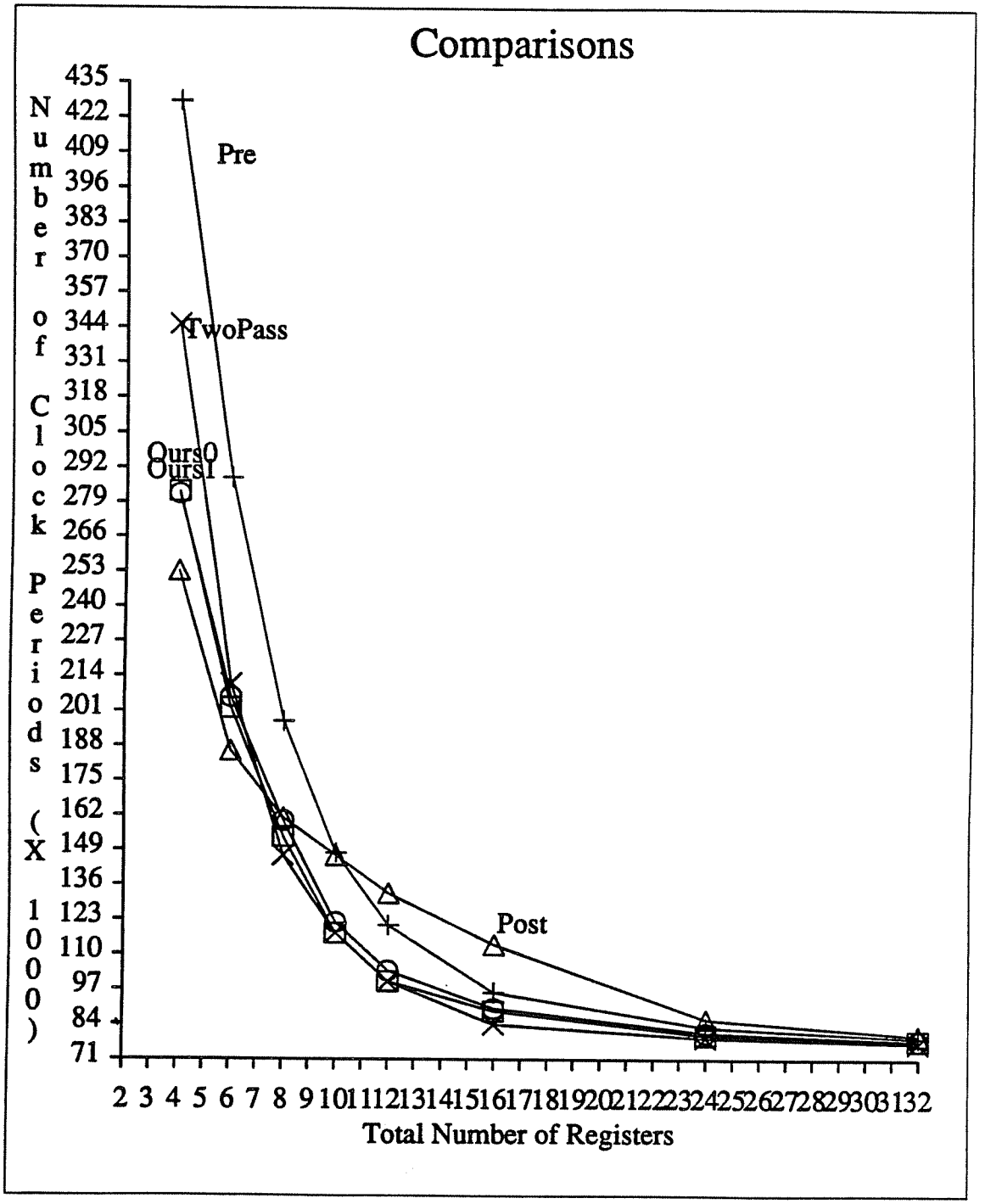


Figure 5.19 Comparisons Using Total Number of Registers (Medium Pipelined)

In Figure 5.18 and Figure 5.19, the total number of registers is used for the x axis. If we want to compare Figure 5.18 and 5.19 with Figure 5.12 and 5.13, a fair way is to look at the curves in Figure 5.18 and 5.19 with the total number of registers more than eight. Since each loop needs three to ten allocated registers (for frequently used variables and constants; loop eight needs 20), four Available Registers (the starting number on the x axis in Figure 5.12 and 5.13) implies 7 to 14 total registers. As in Figure 5.12 and 5.13, prepass scheduling usually outperforms postpass scheduling and the algorithm Ours0 and Ours1 are better than either prepass or postpass schedulings. Twopass scheduling has slightly better performance than Ours0 and Ours1. In fact, twopass scheduling has about the same performance as Ours0 and Ours1 except for loop eight where twopass scheduling significantly outperforms Ours0 and Ours1. Since loop eight needs 20 registers allocated for variables and constants, the total number of registers is not enough for many points in Figure 5.18 and 5.19. How many of the 20 allocated pseudo-registers should be spilled in order to facilitate code scheduling in reducing pipeline interlocks? This is really dependent on how frequently and where the allocated pseudo-register are used. Obviously, our profitable spilling algorithm is a little conservative for such a loop so that Twopass scheduling has better performance. However, as we discussed before, the operand fetch delay due to limited memory bandwidth has not been taken into account. Twopass scheduling generates many more loads and stores which will be a serious problem when the memory bandwidth is the performance bottleneck.

5.6.2.3. Problems with Multiple Functional Pipelines

The model architecture that we choose has parallel function units as shown in Figure 5.11. Two problems should be discussed with such an architecture. The first problem is the result bus conflict. Since any two functional units may finish at the same time, the conflict in using the result bus should be solved. This problem can be solved in hardware by using a "result shift register" to control the result bus [Smit85]. Software solution is also possible by inserting "NOP" when a result bus conflict is detected at compile time. The second problem is the imprecise interrupts [Smit85]. Suppose instruction i takes more cycles to finish than instruction j does. If instruction j is issued later than instruction i but finishes earlier, then an exception condition which occurs after j's termination but before i's termination may leave the processor in an undefined state.

A natural way to avoid the above problems is to assume function units having the same length. This is similar to a linear pipeline implementation [Smit85]. For those operations that can be finished early in the pipe, they still proceed to the end of the pipe so that instructions update the register file in their issue order. The major penalty of this implementation is the decreased instruction issue rate. Since instructions waiting for results from simple operations can not be issued until the results propagate through the pipe and update the register file. In theory, if there is enough parallelism, those idle slots can be filled up with independent instructions by the compiler. Figures 5.20 and 5.21 are based on a linear pipeline implementation. Since the average loop size is about 32, which does not have sufficient parallelism, it takes many more cycles to execute (compared to Figure 5.12 and 5.13).

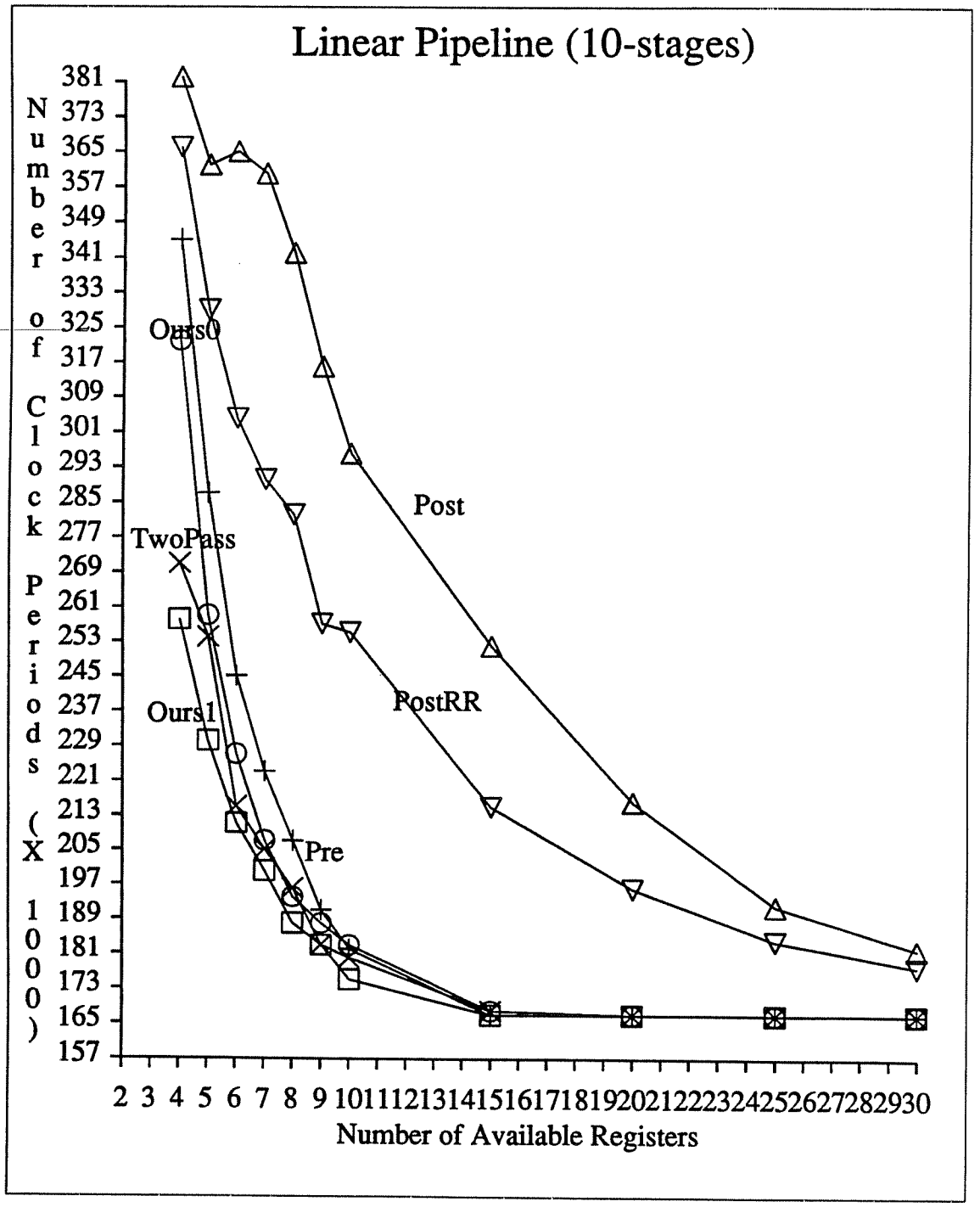


Figure 5.20 Linear Pipelined Model (10-stages)

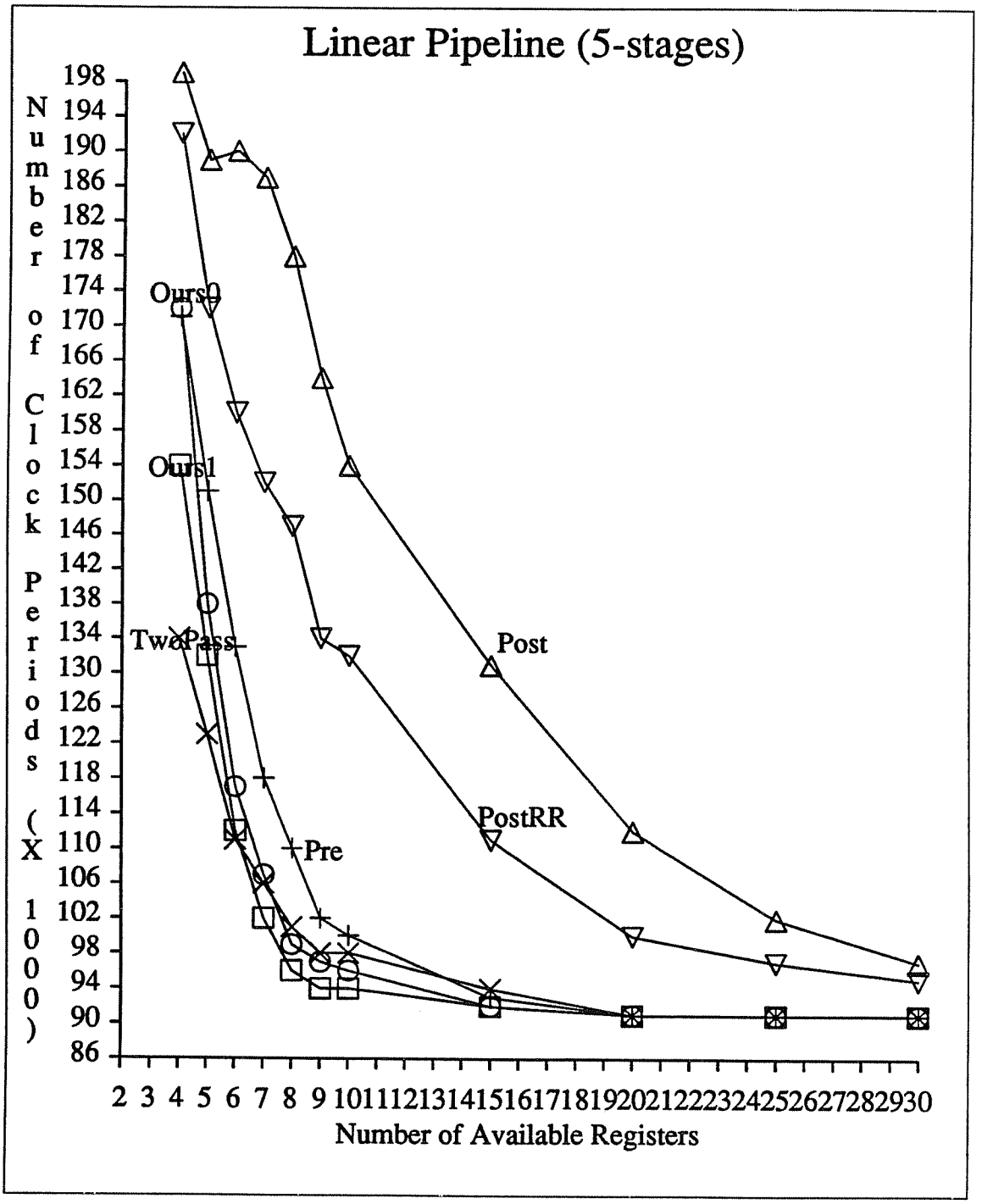


Figure 5.21 Linear Pipelined Model (5-stages)

In Figure 5.12 and 5.13, Prepass scheduling has worse performance than Postpass scheduling when the number of available registers is very small. This is because Prepass scheduling generates lots of loads and stores for spilling and the load instructions may cause long interlocks with instructions which subsequently use the operand. In Figure 5.20 and 5.21, it is interesting to note that in the linear pipelined model, Prepass has better performance than Postpass scheduling even when the number of available registers is small. In the linear pipeline model, a load instruction takes the same number of cycles as other instructions while in our previous model a load instruction takes much longer than other instructions. Therefore, spilling is relatively less expensive in the linear pipeline model. Thus, spilling is advantageous in reducing interlocks in a machine model where memory access operations are faster than other register-to-register operations. Usually, memory operations are much slower than register-to-register operations. However, if there is an appropriate memory hierarchy, memory latency can be hidden effectively. For example, the Cray-1 architecture has a B and T register files, which can be treated as local memory. Data commuting between A and B register files (or S and T) is faster than other operations. Therefore, spilling A (or S) registers to B (or T) registers is favorable if long pipeline interlocks are encountered at compile time.

The algorithm Ours1 which considers profitable spilling outperforms Twopass scheduling for the machine model with 10 pipeline stages as shown in Figure 5.20. However, Ours1 is inferior to Twopass for the machine model with five stages as shown in Figure 5.21. This is not because Twopass works better with less pipelined machine models. This reveals that the algorithm Ours1 is not robust over different machine models. As we described before, when Ours1 is issuing instructions from CSR phase

and encounters an interlock greater than a threshold, it will revert to CSP. The threshold value should be varied with different parameters. In the current implementation, this value is a constant; therefore, Ours1 does not spill as much in the machine model with five stages as in the machine model with 10 stages, since a 10 stage pipeline model is more likely to have long interlocks compared with a five stage pipeline model.

For results to be used early, bypass paths may be provided from immediate pipeline stages to the register file output latches, see Figure 5.22. The added bypass network can significantly improve the instruction issue rate especially when most instructions can be finished in a small number of clock periods. However, bypass paths are complex, expensive, and likely to slow down the clock rate. More importantly, bypassing creates a difficult problem for code scheduling: if the code scheduler knows there is a bypass path, then perhaps it should schedule the dependent instructions close to each other rather than separate them away [Band87]. This is similar to the "chaining" property of vector instructions on Cray-1 like machines [Cray82, Arya85, Bem86]. *Synchronous chaining* refers to the immediate transfer of data between pipelines involved in successive vector instructions. Such chaining requires that the second instruction be prepared to issue at the time that the first result emerges from a pipeline. The time at which the first result emerges is called "chain slot" time. Failure to chain requires that the first vector instruction be carried to completion; when the result vector is completely stored in a vector register, the second instruction can issue. Analogously, if an instruction misses the bypassing time then the instruction can not issue until the previous instruction which computes the result proceeds to the end of the pipeline and updates the register file. Therefore, the code scheduler can either schedule dependent

instructions close to each other to catch up data bypassing or schedule them far away from each other to avoid interlocks.

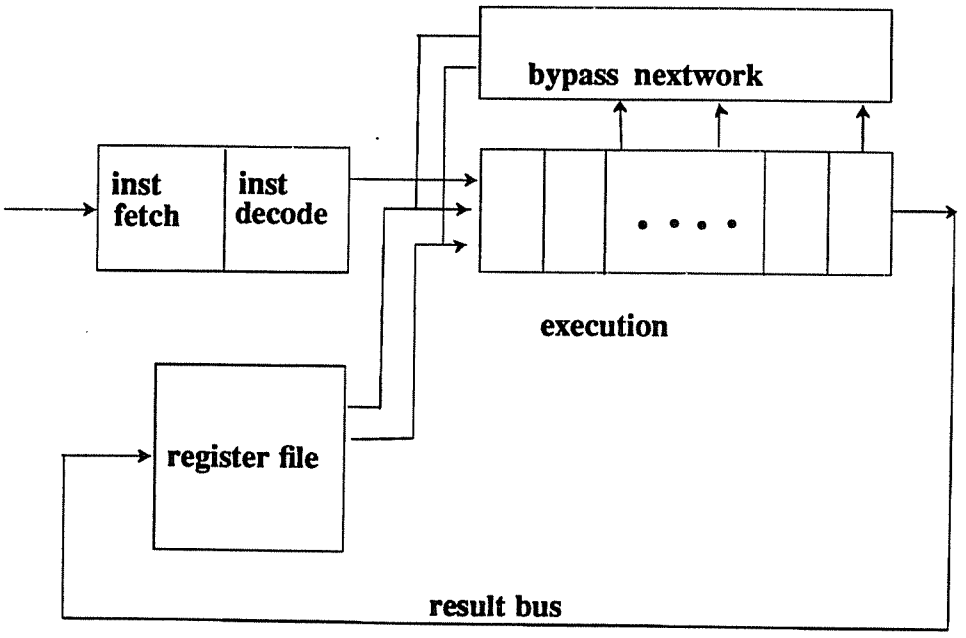


Figure 5.22 Bypass Network

Chapter 6
DAG-Driven Register Allocation

6.1. Motivation

In the previous chapter, we suggested solving the interdependency problem using integrated prepass scheduling. In this chapter, we study an alternative approach which is for the postpass scheduling. This approach uses the dependency DAG to minimize extra dependencies added by register allocator during register allocation. If the DAG-driven register allocator introduces few extra dependencies, the postpass code scheduling will not suffer great performance degradation.

There are three advantages to performing code scheduling in a postpass scheduler. First, the postpass scheduler can be applied both to code output from a compiler and to hand-written assembly-language code [Henn83]. However, programmers today seldom use assembly language except where optimal performance is essential or machine functions are required that are not reflected in the source language. If a compiler can produce object code that is close to the best hand code (for example, the PL.8 compiler), the necessity for assembly language programming is miniscule. Therefore, this advantage seems less important than the other two.

Second, postpass scheduling will never introduce any additional spill code, since register allocation has been done. Although the integrated scheduling method that we proposed in the previous chapter is effective in controlling register spilling, it still generates some spill code as shown in Figure 5.13 and 5.14. If minimizing memory traffic is the major concern, then postpass scheduling should be favored. DAG-driven register allocation is a way to improve the performance of the postpass scheduling

approach.

Last, some architectures have difficulty problems with prepass scheduling. For example, in the WISQ architecture [Ples87], software branch prediction is used in order to schedule code across basic blocks. When branch prediction turns out to be wrong at run time, some of the instructions that are moved from the predicted target block may have to be undone. This is realized by setting a bit mask, which indicates what instructions should be nullified, associated with the branch instruction. If prepass scheduling is used, the bit masks will be set after code scheduling. Later in the register allocation, some inserted spill code may invalidate the bit masks and require a rescheduling to make the bit masks correct. Postpass scheduling avoids such an awkward situation.

6.2. Balancing DAG Reconstruction

We define the *width* of the DAG as the maximal number of mutually independent nodes which need a destination register (a store instruction, for example, does not need a destination register), and the *height* of the DAG as the length of the longest path of the DAG. Since we use the single assignment rule in naming temporaries, the dependency DAG will have a maximal width which exposes maximal parallelism. If the number of real registers is larger than the width of the DAG, the shape of the DAG can remain unchanged during register allocation. Otherwise, the register allocator will reduce the width of the DAG to be smaller than or equal to the number of real registers by reusing registers. While the width is reduced, the height is increased since each reuse of registers may merge two evaluation paths into one. The greater the height, the longer the critical path. The longer the critical path, the less efficient the code

scheduling. Therefore, the goal of our DAG-driven register allocator is to minimize the height of the reconstructed DAG. Two strategies to control the growth of the height of the reconstructed DAG include exploiting free WAR dependencies and balancing the growth of the DAG.

6.2.1. Free WAR Dependencies

The reuse of a register creates new dependencies, primarily write-after-read (WAR) dependencies. We have explained that the added WAR dependencies reduce available parallelism and result in less effective code scheduling in chapter 5. We assumed a pipeline structure in which the operand registers are read at the time an instruction is issued (see Figure 5.11). So long as instructions issue in order (at run time), WAR hazards at register level will never occur. Therefore, the WAR dependency edges are essentially used to enforce the logical order of instructions. We assign WAR dependency a cost of 1, the lowest cost of all dependencies.

Figure 6.1 shows the DAG of the following program segment. Assume there are 5 registers. As the register allocator reads the program, it allocates register R1 through R5 for the destination registers of instruction 1 to instruction 5. At instruction 6, since the available registers have been used up, the register allocator tries to find a dead register to replace. Two registers, register 2 and register 4, are dead before instruction 6. As in Figure 6.1, the reuse of register 4 at instruction 6 introduces a WAR dependency which is represented as a dashed line from instruction node 5 to 6. This dependency is redundant since the logical order of instructions 4, 5 and 6 can be enforced by existing dependency edges. But if register 2 is allocated rather than register 4 to instruction 6, the added dependency (from node 3 to node 6) is not redundant. Redundant

```
1 Load PR1, a
2 Load PR2, b
3 Add  PR3, PR1, PR2
4 Load PR4, c
5 Sub  PR5, PR4, PR1
6 Addi PR6, PR5, #4
7 Mul  PR7, PR5, PR1
.
.
```

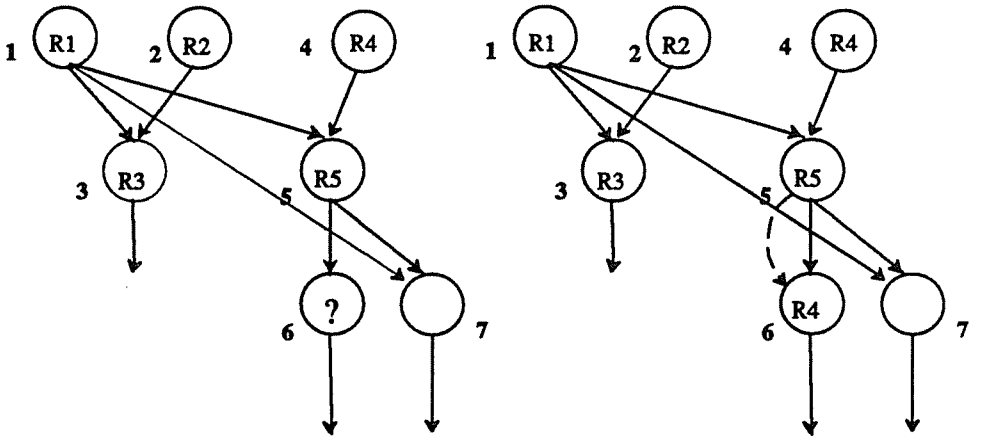


Figure 6.1 Free WAR Dependencies

dependencies are *free dependencies*, since they will not increase the height of the DAG.

To minimize the increase in height of the DAG, the register allocator will first select a dead register to replace such that only redundant dependencies are introduced. In other words, the uses of the dead register are on the dependent path of the current instruction. For example, in Figure 6.1, when the register allocator allocates register 4 for instruction 6, the last use of register 4 is instruction 5, which is on the dependent path of instruction 6. Therefore, this allocation introduces no additional dependencies.

6.2.2. Balancing the Growth of the DAG

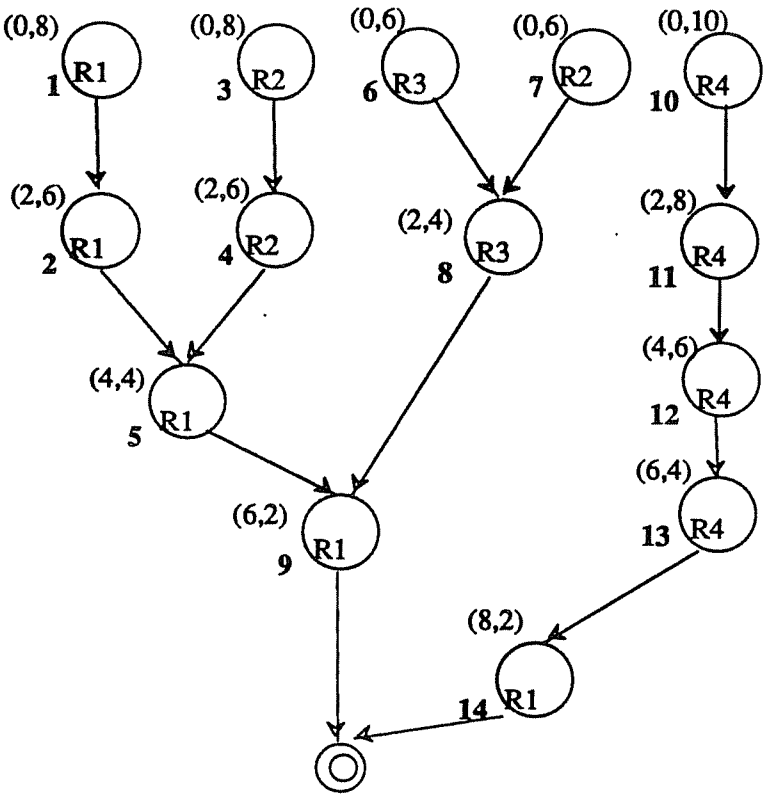
All register replacements which add new dependencies may increase the height of the DAG. When there are no free dependencies, we allocate registers based on *earliest issue time* and *earliest finish time*. The earliest issue time (EIT for short) of a node is the maximal path cost from the beginning of the DAG to the node. The earliest finish time (EFT for short) of a node is the maximal path cost from the node to the end of the DAG. As the names suggest, the EIT of a node indicates the earliest possible issue time of that instruction from the beginning of the execution of the DAG. The EFT of a node indicates the earliest possible finish time from the issue of that instruction to the end of the execution of the DAG. Figure 6.2 shows the EIT and EFT attributes of each node in a DAG. If the register allocator assigns two independent paths to share a register, then a new WAR dependency connects the two paths into one long path. For example, in Figure 6.1, if the register allocator assigns register 2 to instruction 6, then instruction node 6 is connected with instruction node 3 by a WAR dependency edge. The maximal cost of this new path is apparently $(EIT_3 + EFI_6 + 1)$, where 1 is the cost of the WAR dependency edge.

Intuitively, it seems that selecting a node which has a minimal EIT for connection will minimize the growth of the height of the DAG. For example, in Figure 6.3, when instruction node 10 is processed, there are three dead registers: registers 2, 3 and 4. Replacing register 4 seems a good choice because apparently node 8 has a smaller EIT than node 5 and node 9 have. That is, connecting node 10 to node 8 minimizes the increase of the height. This is not always true. Let us look at Figure 6.4, which is the same as Figure 6.3 except that there is one more path, which is even longer than the path headed by node 10. If the allocator assigns register 4 to node 10, then node 15 is

forced to pick register 2 or register 3. This allocation will result in a longer height than an allocation which assigns register 3 to node 10 and register 4 to node 15.

The key idea in minimizing the height of the DAG is to balance the growth of the DAG. The allocator tries not to connect two nodes such that one has a large EIT and the other has a large EFT. If the current instruction has a high EFT, then the allocator would select a dead register such that all the nodes the current instruction will connect to have a small EIT. But how does the allocator know if the EFT of the current instruction is relatively large or small? This suggests that the allocator should look at all the unallocated instructions, especially the leader nodes, to determine where the current instruction stands.

Statically computed EIT cannot be used directly in allocating registers since each register replacement may change the EIT of some nodes. We will talk about the implementation in the following section.



numbers in parenthesis are (EIT, EFT)
assuming each instruction takes 2 clock periods
the double circle is a pseudo node which
indicates the end of the DAG

Figure 6.2 Computations of EIT and EFT

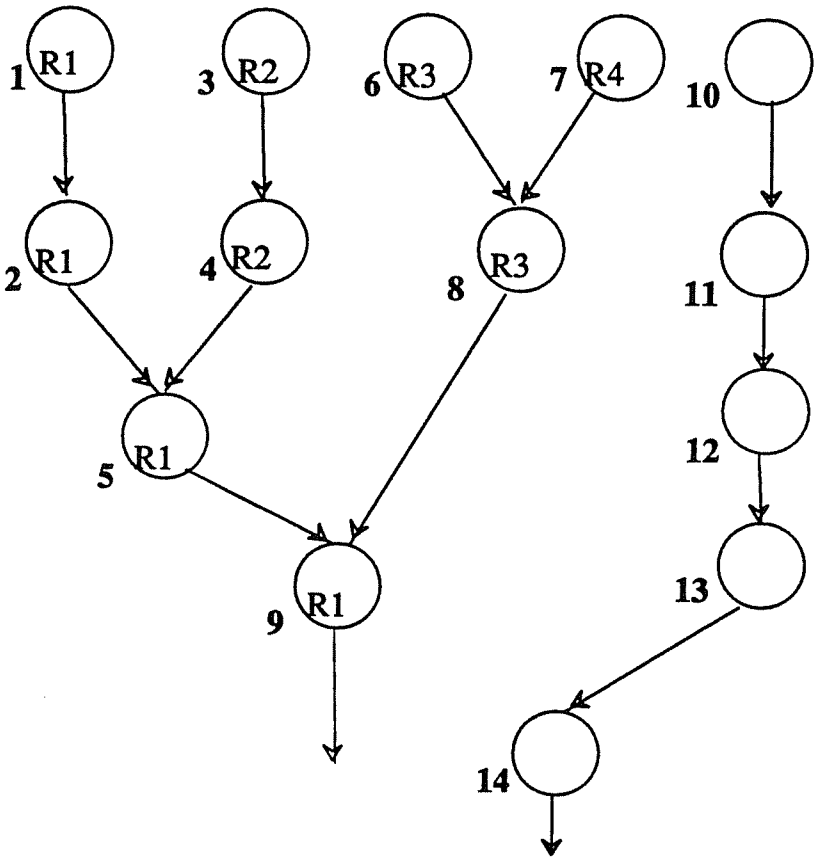


Figure 6.3 Example DAG

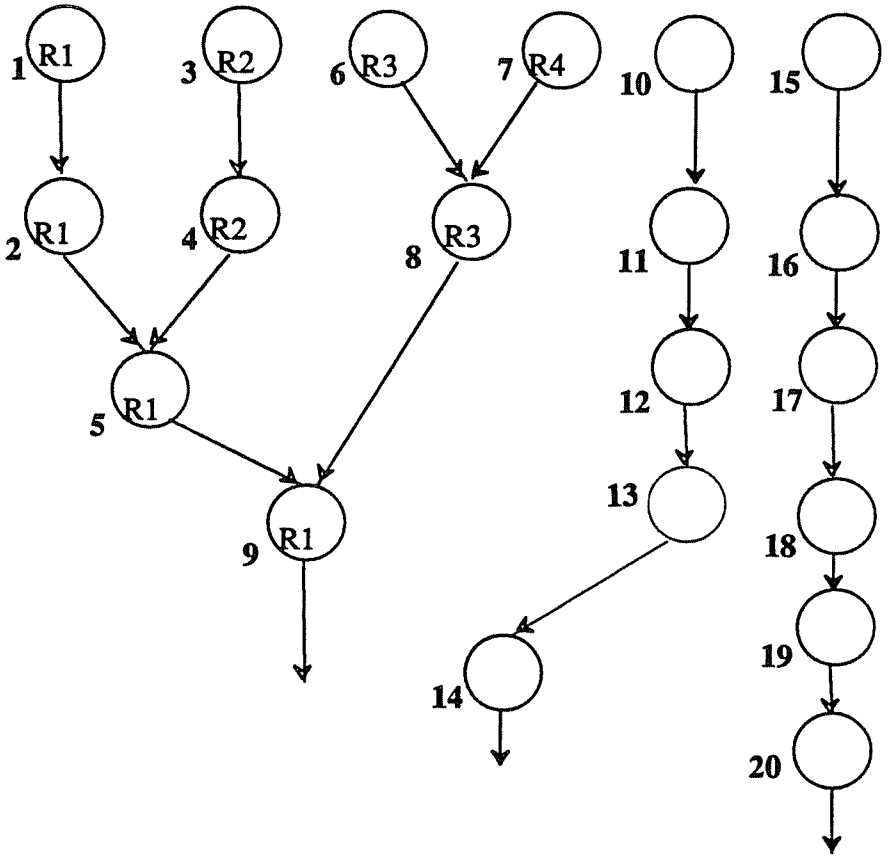


Figure 6.4 Another Example DAG

6.3. Implementation Notes

6.3.1. Update Dependent Relation Incrementally

Our first strategy is to select a dead register whose last uses (a pseudo-register may be read several times, *i.e.*, a common sub-expression) are on instruction nodes on which the current instruction depends (hence, the added WAR dependencies are all free dependencies). To quickly answer questions like "does instruction N depend on instruction M?", we represent the dependency relation of instructions by an adjacency

matrix and perform a transitive closure operation on it [Sedg83]. This is not enough, however, since the dependent relation is changing incrementally during register allocation. For example, suppose instruction N does not depend on instruction M in the original DAG. But when the allocator assigns a register which is used by instruction M to instruction N, then instruction N depends on both instruction M and all the instructions on which M depends. In addition, all instructions that depend on N now also depend on both M and all the instructions that M depends on. The adjacency matrix should be updated incrementally after every register replacement when the added dependencies are not redundant.

6.3.2. Update EIT Dynamically

The EIT of each node also changes dynamically due to register allocation. In our implementation, we associate the EIT attribute with each real register. The EIT of each real register is updated dynamically and is used to determine which dead register is most appropriate to assign to the current instruction.

We have defined EIT and EFT in the previous section but have not explained how to compute them. We briefly describe the computation of EIT and EFT as follows. We denote the EIT of a node N by EIT_N, the EIT of a register R_i by EIT_{Ri}, the EFT of a node N by EFT_N, the execution time estimate of an instruction I by T_I, and the cost of a dependency edge which leads from node I to node J by T_{IJ}. Initial leader nodes have EIT of 0. Suppose there are three instructions, A, B, and C, such that C depends on A and B. Then EIT_C is computed by $\mathbf{max}(EIT_A + T_{AC}, EIT_B + T_{BC})$. The computation of EFT is similar to that of EIT. Consider three instructions A, B, and C, such that B and C depend on A. Then EFT_A is equal to $\mathbf{max}(EFT_B + T_{AB}, EFT_C + T_{AC})$. There is a

pseudo node in the DAG indicating the end of the DAG. All end nodes (the nodes that no nodes depend on except the pseudo node) have a dependency edge lead to the pseudo node. The EFT of the pseudo node is 0.

There are several attributes for each real register: the EIT field, the T field (execution time estimate of the latest write operation), a variable length list of new EIT values (called E-list), and some information for handling register replacements (*e.g.* dirty or clean). All fields are initialized to 0 or appropriate values. Suppose an instruction I reads real register j and k, and writes register i (i, j, and k are assigned by the register allocator). The attributes are updated as follows:

(1) $EIT_I = \max (EIT_{Rj} + T_{Rj}, EIT_{Rk} + T_{Rk})$

(2) Attach the value of (EIT_I) to the end of the E-list of register Rj. If this is the last read of register Rj, select the maximal value from the E-list and assign it to EIT_{Rj}, and then clear the E-list of Rj.

(3) Attach the value of (EIT_I) to the end of the E-list of register Rk. If this is the last read of register Rk, select the maximal value from the E-list and assign it to EIT_{Rk}, and then clear the E-list of Rk.

(4) If one of the last uses of register Ri is on a path which instruction I does not depend on, then $EIT_{Ri} = \max (EIT_I, EIT_{Ri} + 1)$. Otherwise $EIT_{Ri} = EIT_I$

(5) $T_{Ri} = T_I$

6.3.3. Replacing Live Registers

If there are no dead registers available, a live register is selected. Spill code may be inserted depending on whether the live register is clean or dirty. For a read miss, a load instruction is inserted to fetch the operand. Figure 6.5 illustrates these cases. If register Ri is spilled, the maximal value from its E-list is assigned to EIT_{Ri}. Then update EIT_{Ri} by $EIT_{Ri} + T_{store}$. If there is a read miss and the register assigned for the load instruction is register Ri, EIT_{Ri} is updated by $EIT_{Ri} + T_{load}$.

6.3.4. The Algorithm

The following is an outline of the allocation algorithm

(1) Rename pseudo-registers to enforce single assignment

(2) Use CSR to reduce the number of simultaneously live registers

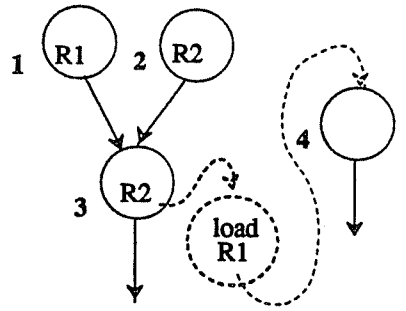
(3) Read in the basic block, build the DAG and set up a hash table of register reference histories

(4) Compute the EFT of each node, set up the adjacency matrix representing the dependency relation

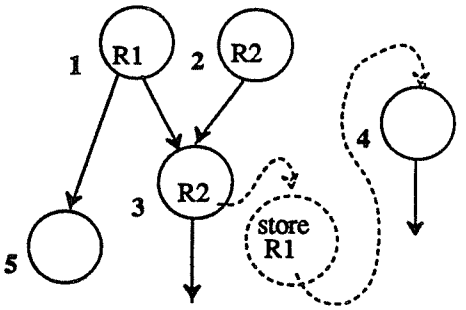
(5) Register allocation:

while there are instructions to be allocated **do**

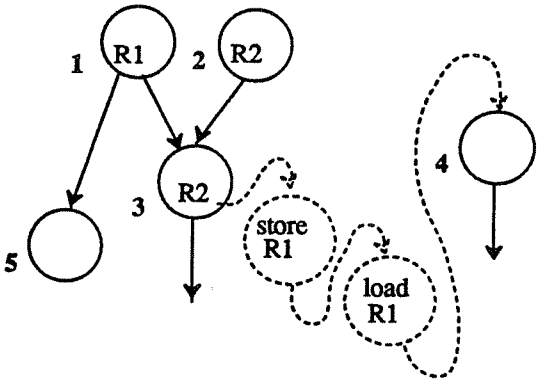
foreach pseudo-register in the instruction, in the order of
first operand, second operand, destination **do**



node 4 involves a read miss
register R1 is dead after node 3



node 4 involves a write miss and
live register R1 is spilled



node 4 involves a read miss and
live register R1 is spilled

Figure 6.5 Complications due to Spilling

```
if (pseudo-register Miss) then

   if (Write Miss) then

      if (there are dead registers on the dependent path) then
          select one and replace;
          go to 2;
      endif
   endif
   if (there are dead registers) then
```

```
            sort them in ascending order on the EIT field,
            count how many remaining leaders have a higher EFT
              than the current instruction into a variable P,
            if (P+1 is greater than the number of dead registers) then
              select the dead register with highest EIT to replace;
            else
              select the P+1st dead register to replace;
            endif
          else
            normal replacements;
          endif
        endif
      endfor

      1: update dependent relation incrementally;

      2: update EIT of real registers;

    endwhile
```

6.3.5. Example

The following example serves to illustrate how EIT's are computed and how registers are selected for replacement. The sample program is the same as in Figure 5.1. The IL program is first scheduled using CSR to reduce the number of simultaneously live registers. The code sequence after the scheduling is in Figure 5.4. For the convenience of references, the code sequence is duplicated here.

```
1 Load PR4, c
2 Load PR5, d
3 Add  PR6, PR4, PR5
4 Load PR1, a
5 Load PR7, e
6 Add  PR8, PR1, PR7
7 Mul  PR9, PR8, PR6
8 Load PR2, b
9 Mul  PR3, PR1, PR2
10 Add  PR10, PR3, PR9
11 Stor PR10, h
```

The assumption of the timings are the same as before. That is, a load takes 4 clock periods, an add takes 2, a multiplication takes 3, and a store takes 1. Following step 3 and 4 in the algorithm, we construct the DAG and compute the EFT shown in Figure 6.6.

Assume there are four real registers. Contents of register attributes will be displayed step by step.

Step 1: a write miss for PR4. After assigning register 1 for PR4, the contents of attributes are:

register	R1	R2	R3	R4
tag	PR4	invalid	invalid	invalid
EIT	0	0	0	0
T	4	0	0	0
E-list	empty	empty	empty	empty

Step 2: a write miss for PR5; register 2 is assigned to PR5.

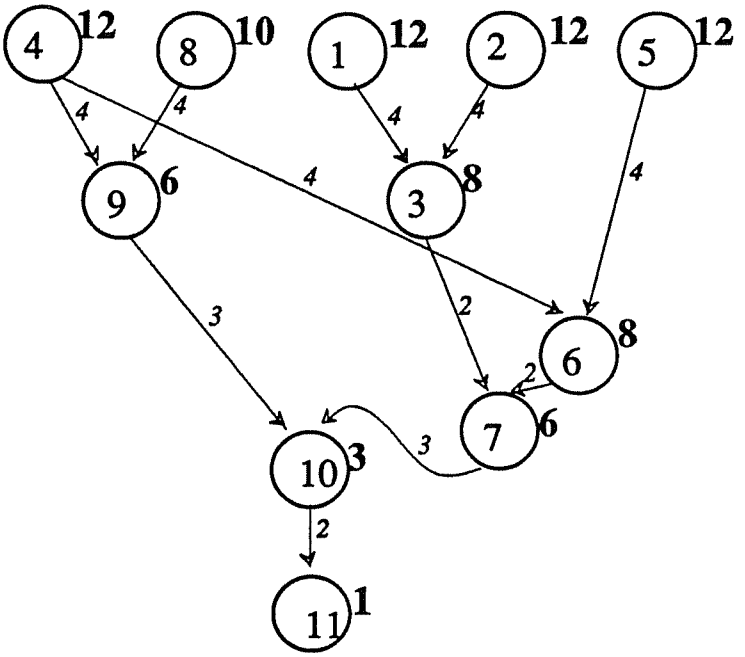


Figure 6.6 WDAG with Computed EFT

register	R1	R2	R3	R4
tag	PR4	PR5	invalid	invalid
EIT	0	0	0	0
T	4	4	0	0
E-list	empty	empty	empty	empty

Step 3: read hits for PR4 and PR5; a write miss for PR6. Both register 1, which holds PR4, and register 2, which holds PR5, are dead and on the dependent path of instruction 3. Register 1 is selected for replacement. $EIT_3 = \mathbf{max}(EIT_{R1} + T_{R1}, EIT_{R2} + T_{R2}) = 4$. The EIT_3 is attached to the E-list of register 1 and 2. Since instruction 3 is the last read of both PR4 and PR5, the E-list of register 1 and register 2 will be cleared after the

processing of instruction 3.

register	R1	R2	R3	R4
tag	PR6	PR5	invalid	invalid
EIT	4	4	0	0
T	2	4	0	0
E-list	empty	empty	empty	empty

Step 4: a write miss for PR1. No remaining leaders have a higher EFT than instruction 4. Therefore, register 3, which has a lowest EIT, is assigned to PR1.

register	R1	R2	R3	R4
tag	PR6	PR5	PR1	invalid
EIT	4	4	0	0
T	2	4	4	0
E-list	empty	empty	empty	empty

Step 5: a write miss for PR7; register 4 is replaced.

register	R1	R2	R3	R4
tag	PR6	PR5	PR1	PR7
EIT	4	4	0	0
T	2	4	4	4
E-list	empty	empty	empty	empty

Step 6: read hits for PR1 and PR7; a write miss for PR8. Since register 4, which holds PR7, is dead and on the dependent path of instruction 6, register 4 is selected for replacement.

register	R1	R2	R3	R4
tag	PR6	PR5	PR1	PR8
EIT	4	4	4	4
T	2	4	4	2
E-list	empty	empty	(4)	empty

Step 7: read hits for PR6 and PR8; a write miss for PR9; register 1 is replaced.

register	R1	R2	R3	R4
tag	PR9	PR5	PR1	PR8
EIT	6	4	4	6
T	3	4	4	2
E-list	empty	empty	(4)	empty

Step 8: a write miss for PR2. Both register 2 and 4 are dead. They are not on the dependent path of instruction 8. Since there are no remaining leaders having a higher EFT than that of instruction 8, register 2, which has a smaller EIT, is selected to replace.

register	R1	R2	R3	R4
tag	PR9	PR2	PR1	PR8
EIT	6	4	4	6
T	3	4	4	2
E-list	empty	empty	(4)	empty

Step 9: read hits for PR1 and PR2; a write miss for PR9; register 3 is replaced.

register	R1	R2	R3	R4
tag	PR9	PR2	PR3	PR8
EIT	6	4	4	6
T	3	4	3	2
E-list	empty	empty	empty	empty

Step 10: read hits for PR3 and PR9; a write miss for PR10; register 1 is replaced.

register	R1	R2	R3	R4
tag	PR10	PR2	PR3	PR8
EIT	9	4	9	6
T	2	4	3	2
E-list	empty	empty	empty	empty

Step 11: a read hit for PR10.

The program after the above register allocation is shown as follows:

```
Load    R1, c
Load    R2, d
Add     R1, R1, R2
Load    R3, a
Load    R4, e
Add     R4, R3, R4
Mul     R1, R4, R1
Load    R2, b
Mul     R3, R3, R2
Add     R1, R3, R1
Stor    R1, h
```

After code scheduling, the program is:

```
Load    R1, c
Load    R2, d
Load    R3, a
Load    R4, e
Add     R1, R1, R2
Load    R2, b
Add     R4, R3, R4
Mul     R1, R4, R1
Mul     R3, R3, R2
Add     R1, R3, R1
Stor    R1, h
```

This code sequence takes only 16 cycles. Compared with code sequences in Figure 5.10, this code sequence has the best performance.

6.4. The Performance of DAG-driven Register Allocation

Our test results, based on the same test environment described in chapter 5, show DAG-driven register allocation significantly improves the performance of postpass code schedulings (cf. Figures 6.7 and 6.8). We also compared DAG-driven register allocation to the integrated prepass scheduling as described in chapter 5. For the highly

pipelined model (Figure 6.9), the integrated prepass scheduling approach that considers profitable spilling slightly outperforms the DAG-driven register allocation approach. However, in the medium pipelined model (Figure 6.10), DAG-driven allocation approach outperforms all the others. We explain our results as follows: for the highly pipelined model, where an interlock could be relatively expensive, spilling may be profitable. Since the integrated prepass scheduling can easily accommodate profitable spilling, it has slightly better performance than the DAG-driven allocation approach. The DAG-driven allocation approach has the advantage that it does not increase the code size at all. Notice that in our prepass scheduling scheme, the switch between CSP and CSR depends on a threshold value. Without an appropriate value, sometimes it is too late to invoke CSR– additional spill code will be generated. The optimal threshold value varies depending on the complexity of the programs. Therefore, even though the prepass scheduling scheme was designed to control spilling, it can not completely avoid any additional spill code. Since DAG-driven allocation introduces no additional spill code, it outperforms the prepass scheduling method for lightly pipelined machine models.

From the experimental results, both the integrated prepass scheduling and the DAG-driven register allocation approaches are effective for solving the problem of the interdependency between code scheduling and register allocation. Conceptually, the DAG-driven register allocation approach is simpler. However, the integrated prepass scheduling is more flexible. For example, if there is a long evaluation path coming late in the code sequence, the prepass scheduler can just schedule this path early so that this path will be allocated a free register during register allocation. Although we can also allow the DAG-driven register allocation to move code when it is necessary, this may

make the allocator too complicated to be implemented effectively.

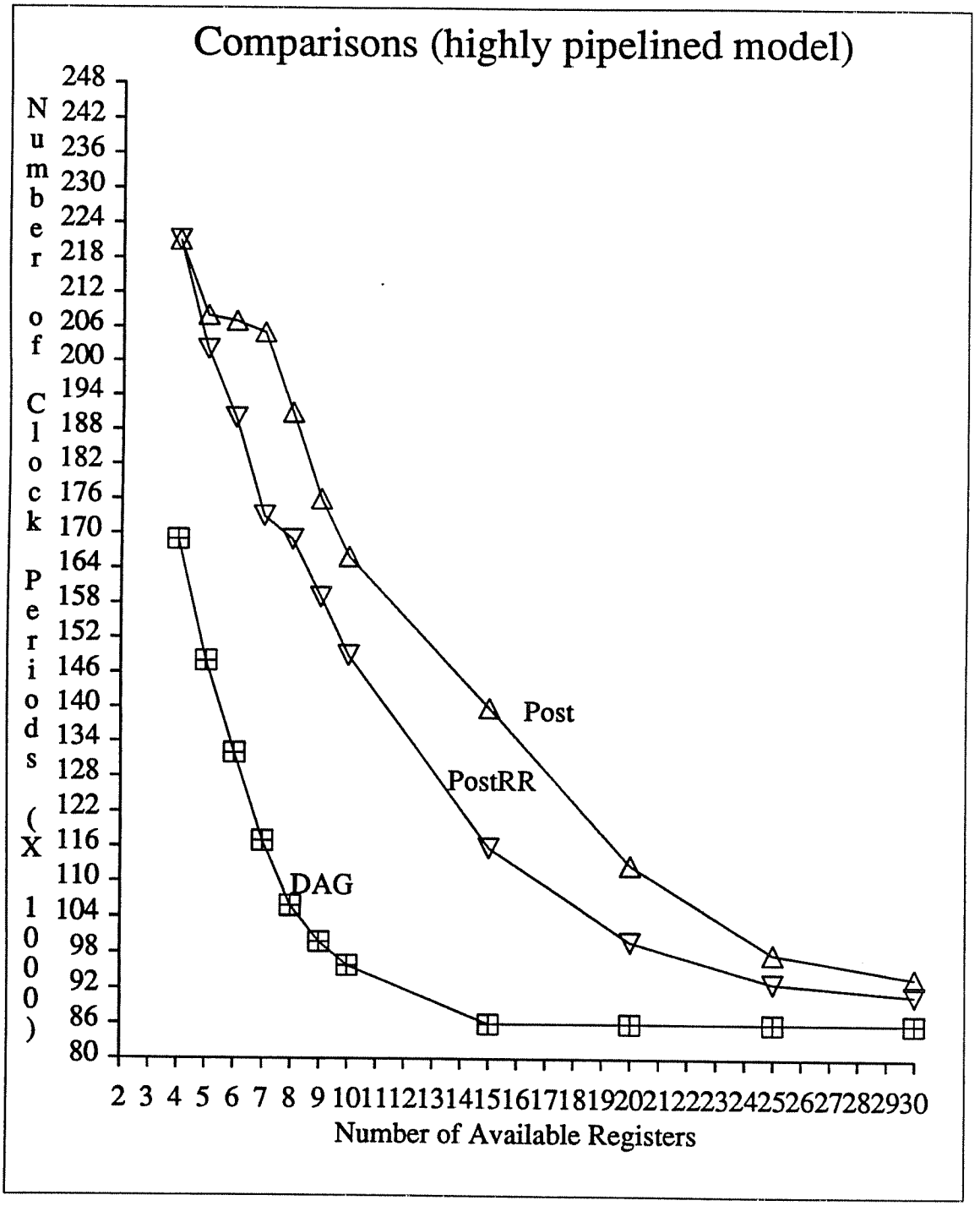


Figure 6.7 DAG-Driven Allocation vs Postpass Scheduling I

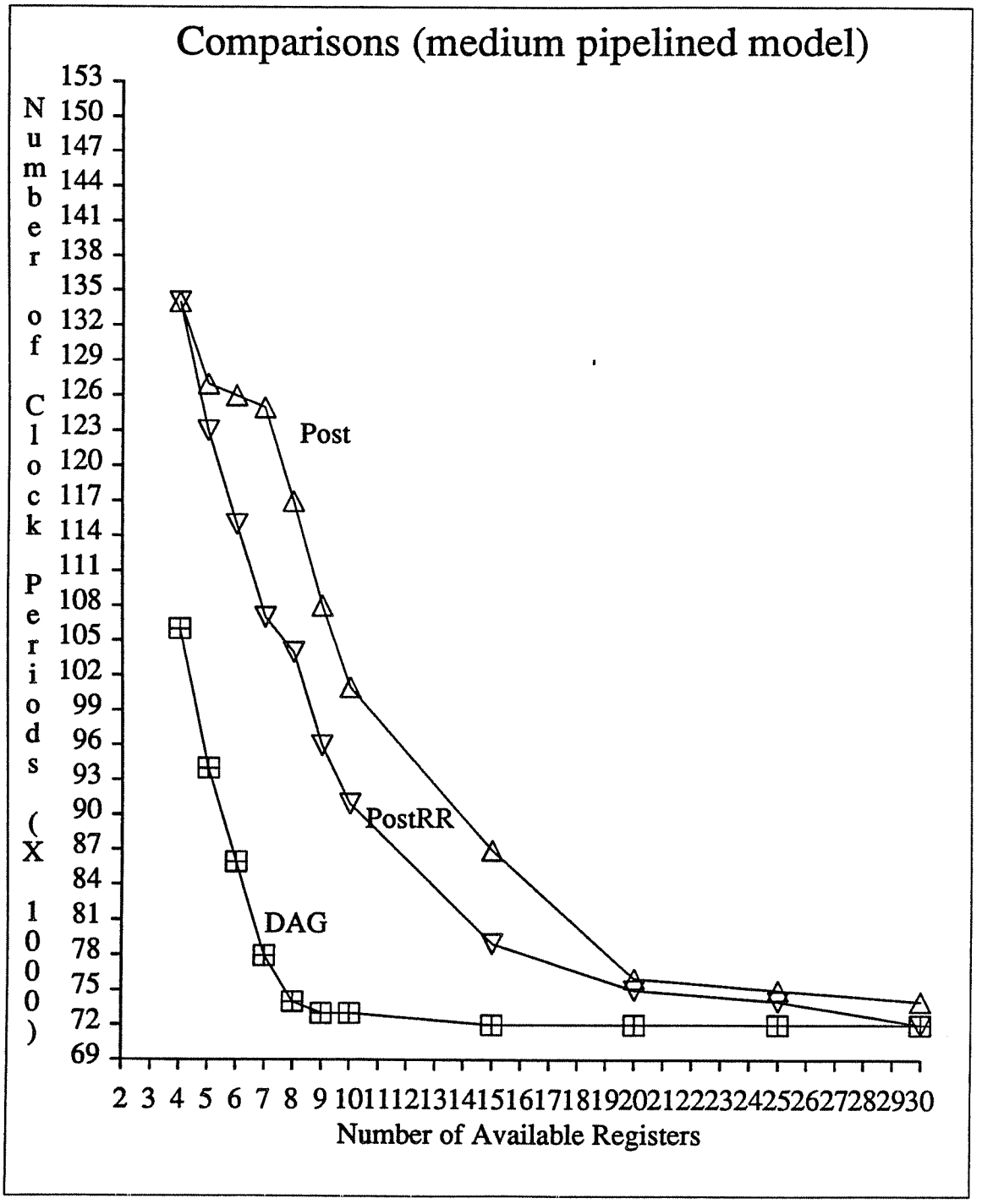


Figure 6.8 DAG-Driven Allocation vs Postpass Scheduling II

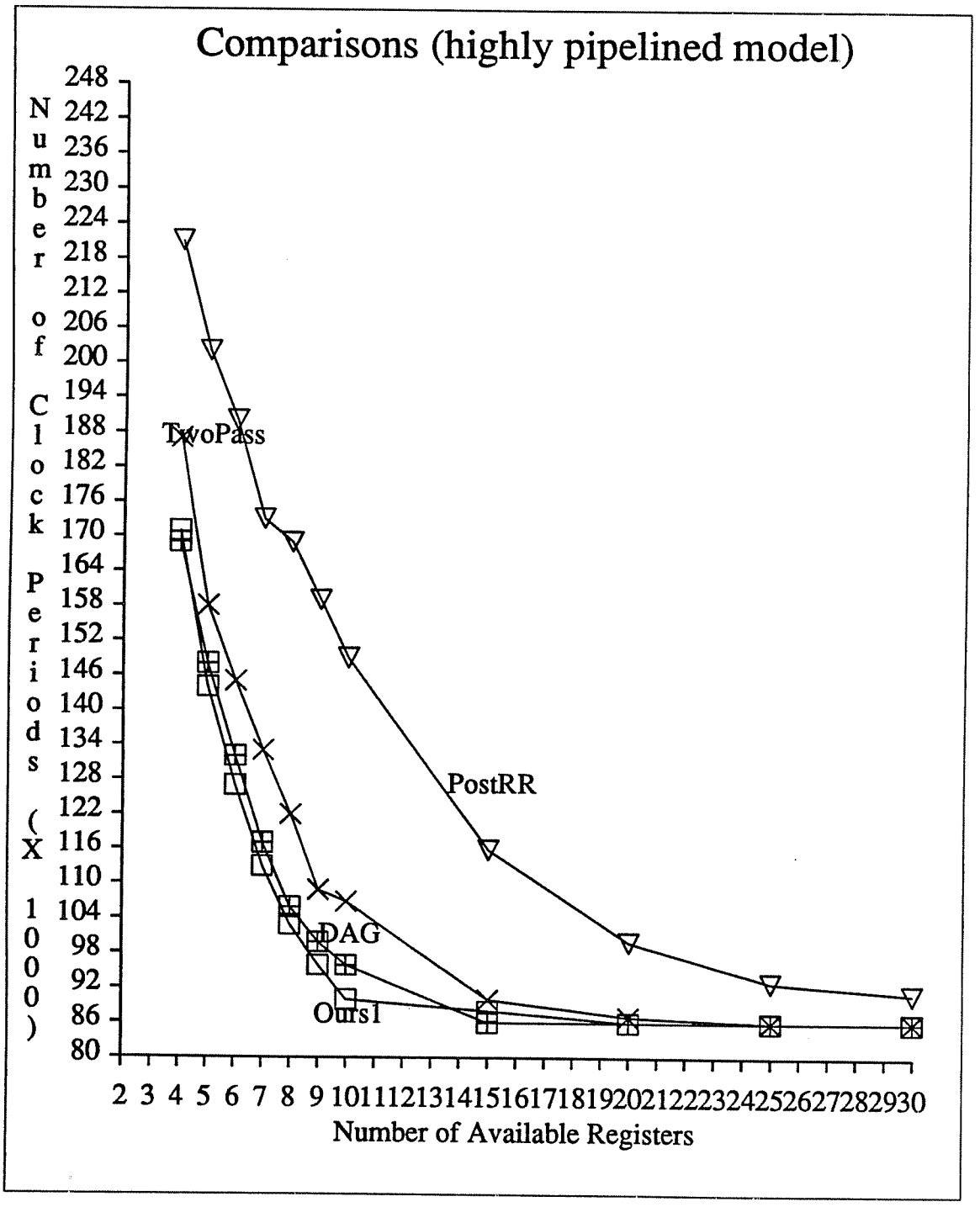


Figure 6.9 DAG-Driven Allocation vs Integrated Prepass Scheduling I

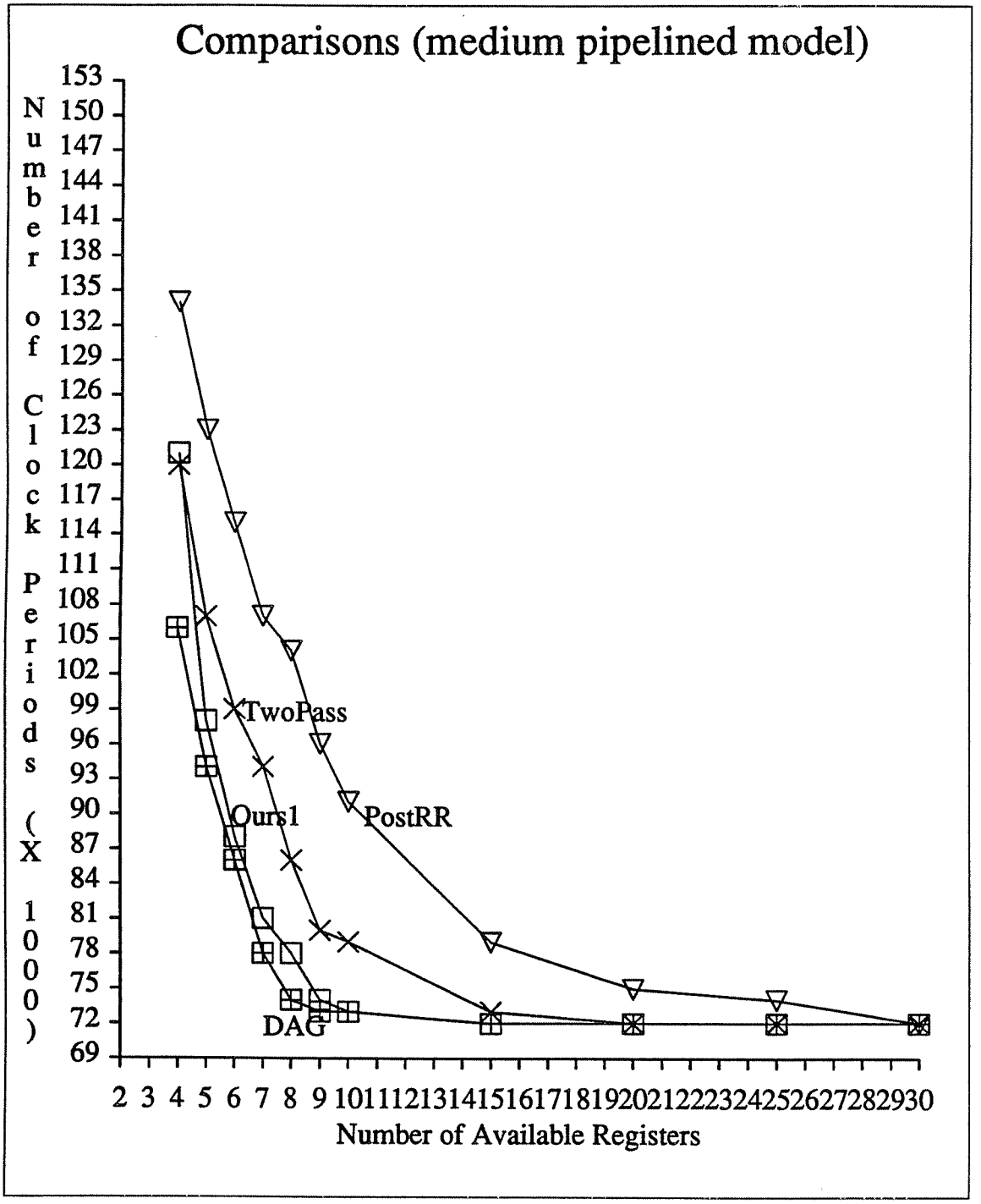


Figure 6.10 DAG-Driven Allocation vs Integrated Prepass Scheduling II

Chapter 7
Conclusions

7.1. Summary of Results

Single-chip computers are becoming increasingly limited by the access constraints to off-chip memory. The off-chip memory latency is long and the off-chip memory bandwidth is limited. To achieve high performance, the structure of on-chip memory must be appropriate, and it must be allocated effectively. Since the off-chip memory bandwidth of single-chip computers is severely limited, on-chip memories that rely on spatial locality to achieve high hit rate are not appropriate. Experimental data presented in chapter 2 has shown that memory accesses of global variables exhibit little temporal locality, except for frequently used scalar variables. Therefore, memory accesses for global variables are unlikely to be supported well by on-chip data caches. A register file, which can be managed by compilers, might be a better candidate as an on-chip memory structure; the compiler can identify dead values to save redundant stores, and can use look-ahead techniques to perform better replacement.

Load/store architectures provide a clean way to use the register file: (1) their nature simplifies the model of register allocation, since minimizing loads and stores minimizes memory traffic; (2) they expose the opportunity to separate load instructions from instructions which use the operands so that memory latency can be hidden. With effective register allocation and code scheduling, load/store architectures can be a nice fit with single-chip computers.

Chapter 3 studied the optimal allocation of registers in basic blocks. The algorithm to find the optimal allocation has been given. The basic approach is to search for a shortest path in a weighted DAG. Since the weighted DAG grows exponentially in the worst case with the number of variables in the input program and the number of available registers, pruning rules have been provided to effectively restrict the worst case to a very small domain. With the provided pruning rules, the algorithm is used to evaluate the effectiveness of existing heuristic algorithms for large basic blocks. An heuristic algorithm which has near-optimal performance and runs in linear time is proposed as a by-product of the optimal algorithm. An extension of the heuristic algorithm to global allocation has also been discussed.

A model to evaluate a perfect register allocation has been proposed in chapter 2. Using the replacement algorithm that we introduced in chapter 3, the performance of the perfect register allocation has been shown to be much more effective in reducing off-chip memory traffic than cache memory of the same size.

Small basic blocks often limit the effectiveness of register allocation and code scheduling. With decreasing memory cost, it may be beneficial to trade code space for more parallelism. Chapter 4 discussed compiler techniques which generate large basic blocks. Those techniques are useful in increasing the effectiveness of parallel/pipelined processing.

Although code scheduling effectively hides memory and function unit latency, it interferes with register allocation, especially in large basic blocks. Chapter 5 and 6 dealt with this problem and presented two methods as solutions: (1) an integrated code scheduling technique; and (2) a DAG-driven register allocator. The integrated code scheduling method combines two scheduling techniques— one to reduce pipeline

delays and the other to minimize register usage— into a single phase. By keeping track of the number of available registers, the scheduler can choose the appropriate scheduling technique to schedule a better code sequence. The DAG-driven register allocator uses a dependency DAG to assist in assigning registers; it introduces much less extra dependency than does an ordinary register allocator. Both approaches were shown to generate more efficient code sequences than conventional techniques in the simulations.

In conclusion, this thesis has

(1) discussed the issues in selecting appropriate on-chip memory structures,

(2) proposed an algorithm for optimal local register allocation for load/store architectures and evaluated how well heuristic algorithms performed for large basic blocks,

(3) shown that register allocation algorithms can potentially reduce memory traffic more effectively than a data cache of the same size,

(4) proposed a new scheduling technique and a new allocation technique to deal with the interdependency between register allocation and code scheduling.

7.2. Suggestions for Future Research

We have discussed the issues of the use and allocation of the register file. Structures other than a register file, like vector registers and queue registers [Youn85] are suitable for array accesses. The allocation of vector registers or queues is not yet well understood, and the scheduling of vector instructions (or queue operations) as well as its relationship to the allocation should be interesting for further study.

Chapter 6 has demonstrated how to reuse registers so that later code scheduling will not be hamstrung by added dependencies. Since the graph coloring register allocation is more widely used than the replacement based register allocation, we are thinking about applying the concept of our work to the graph coloring allocation model. In graph coloring allocation, assigning a color to two nodes is the same as replacing a dead register in our model. Therefore, when more than one color can be assigned to a node, the allocator should be able to select the color which minimizes the added dependency, if the DAG is provided.

While we have shown the advantages of static management of the register file, static management is only suitable for languages that expose most information at compile time. For languages or applications where most of the information can only be determined at run time, caches are certainly a better choice. Tags, which describe the type information of data structures, will be useful in assisting cache management: the cache can use the information described in the tag to perform a more precise prefetch.

Throughout this research, we have assumed that large basic blocks were worth the trouble of increased code space. Although it is generally true that large basic blocks provide better opportunities to exploit parallelism, it is not clear if the increased code space is warranted. While the data accesses may be reduced by more effective use of registers and access latency may be hidden by better code scheduling, more instructions may have to be fetched due to the increased code size. Many researchers believe that instruction caches are more effective than data caches, and therefore are worthwhile in trading instruction bandwidth with data bandwidth. But we have to be careful not to push our approach too far. We should be able to answer a question like: given certain on-chip area, how should we divide this limited memory area for instruction and data.

This certainly requires serious experiments on how effective are techniques for generating large basic blocks, how much more parallelism can be exploited from the increased blocks, how badly the code space will be increased, and how large the instruction cache should be expanded to stay effective.

REFERENCES

[Aho77] Aho, A. V., S. C. Johnson, and J. D. Ullman, "Code Generation for expressions with common subexpressions," J. ACM 24:1, 146-160, 1977

[Aho86] Aho, A. V., J. D. Ullman, and R. Sethi, "Compilers Principles, Techniques, and Tools," Addison-Wesley, Reading, MA, 1986.

[Alle72] Allen, F.E., and J. Cocke, "A Catalogue of Optimizing Transformations," in Rustin[1972].

[Alle80] 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.

[Alle81] Allen, F.E., "The History of Language Processor Technology in IBM," *IBM J. Res. Develop.*, Vol. 25, Sept. 1981.

[Alpe83] 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.

[Arya85] Arya, Siamak, "An Optimal Instruction-Scheduling Model for a Class of Vector Processors," *IEEE Transaction on Computers*, Nov., 1985

[Ausl82] Auslander, M. and M. Hopkins, "An Overview of the PL.8 compiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June, 1982.

[Axel83] Axelrod, T. S., P. F. Dubois, and P.G. Eltgroth, "A Simulator for MIMD Performance Prediction -- Application to the S-1 MkIIa Multiprocessor," *1983 International Conference on Parallel Processing*, Bellaire, MI, pp. 350-357, August, 1983.

[Back57] Backus, J.W., *et al.*, "The FORTRAN Automatic Coding System," 1957, in S. Rosen, editor, *Programming Systems and Languages*, McGraw-Hill, 1967, pp. 29-47.

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

[Band87] Bandyopadhyay, Sumit, V. Begwani, and R. Murray, "Compiling for the CRISP Microprocessor" *Proceedings of the IEEE Spring COMPCON* 1987

[Bela66] Belady, L.A., "A Study of Replacement Algorithms for A Virtual-Storage Computer," "IBM Systems Journal", Vol. 5, No. 2, 1966.

[Bern86] Bernstein, David, Haran Boral, and Ron Pinter, "Optimal Chaining in Expression Trees," *SIGPLAN'86 Symposium on Compiler Construction* June 1986.

[Birn86] Birnbaum, Joel S., and W. Worley, Jr. "Beyond RISC: High-Precision Architecture," *IEEE Spring Compcon Conference*, 1986.

[Chai81] Chaitin, G.J., M. A. Auslander, A. K. Chandra, J. Cocke, M.E. Hopkins and P. W. Markstein, "Register Allocation Via Coloring," *Computer Language*, 6, 1981.

[Chai82] Chaitin, G.J., "Register Allocation and Spilling Via Graph Coloring," *SIGPLAN 82 Symposium on Compiler Construction* June, 1982.

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

[Chow84] Chow, F., and J. L. Hennessy, "Register Allocation by Priority-Based Coloring," *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.* June, 1984

[Clar84] Clark, D., and J. Emer, "A Characterization of Processor Performance in the VAX 11/780", *Proc. 11th Annual Symposium on Computer Architecture*, ACM/IEEE, June 1984.

[Cock70] Cocke J., and J. T. Schwartz, "Programming Languages and Their Compilers," Courant Institute of Mathematical Sciences, NYU, 1970.

[CoutA86] Coutant, Deborah S., Carol L. Hammond, and Jon W. Kelly, "Compilers for the New Generation of Hewlett-Packard Computers," *IEEE Spring Compcon Conference*, 1986.

[CoutB86] Coutant, Deborah S. "Retargetable High-Level Alias Analysis," *Conf. Record of the 13th ACM Symp. on Principles of Programming* Languages," Jan. 1986

[Coop84] Cooper, K. D., "Analyzing Aliasing of Reference Formal Parameters," *Conf. Record of the 11th ACM Symp. on Principle of Programming Languages,* Jan. 1984.

[Cray82] Cray Research Inc., *Cray-1 Computer System* S Series Mainframe Reference Manual (HR-0029), 1982.

[Cray84] Cray Research Inc., *The Cray X-MP Series of Computer Systems*, Technical Brochure, August 1984.

[Davi84] Davidson, Jack and C. Fraser, "Register Allocation and Exhaustive Peephole Optimization," *Software-Practice and Experience*, Sept., 1984

[Davi86] Davidson, J. W., "A Retargetable Instruction Reorganizer," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction.* June, 1986

[Day70] Day, W. H. E., "Compiler Assignment of Data to Registers," *IBM System Journal*, 9(4), pp. 281-317, 1970.

[Dong79] Dongarra, J. J. and A. R. Jinds, "Unrolling Loops in Fortran," *Software Practice and Experience* 9, 3, pp. 219-226, March 1979

[Ditz82] Ditzel, D., and H. McLellan, "Register allocation for free: The C machine stack cache," *Symposium on Architecture Support for Programming Languages and Operating Systems*, 1982.

[Ditz86] Ditzel, D., personal communication.

[Ditz87] Ditzel, D., H. McLellan, and A. Berenbaum "The Hardware Architecture of the CRISP Microprocessor," *The 14th Annual Symposium on Computer Architecture,* June, 1987.

[Elli85] Ellis, J. R., "Bulldog: A Compiler for VLIW Architectures," *Ph.D. Thesis*, YaleU/DCS/RR-364, Yale University, Feb., 1985

[Fisc87] Fischer, C. N. and LeBlanc, "Crafting a Compiler," Benjamin Cummings, 1987.

[Fish80] Fisher, J., "An Effective Packing Method for Use with 2^n-way Jump Instruction" Hardware" 13^{th} Annual Microprogramming Workshop, Colorado Springs, Nov. 1980, SIGMICRO Newsletter, 64-75

[Fish81] Fisher, J., "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981.

[Frei74] Freiburghouse, R.A., "Register allocation via usage counts," *CACM*, 17:11 638-642 Nov., 1974.

[Gibb86] Gibbons P. B., and S. S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction.* June 1986

[Good83] Goodman, J. R., "Using Cache Memory to Reduce Processor/Memory Traffic," *The Tenth Annual Symposium on Computer Architecture,* pp. 124-131, June, 1983.

[Good85] Goodman, J. R., J. Hsieh, K. Liou, A. Pleszkun, P. Schechter, and H. Young, "PIPE: A VLSI Decoupled Architecture" *The 12th Annual International Symposium on Computer Architecture* June 1985.

[Haik84] Haikala, I. J. and Kutvonen, P. H., "Split Cache Organization," Report C-1984-40, Dept. of Computer Science, University of Helsinki, Finland, Aug., 1984.

[Hamm77] D. W. Hammerstrom and E. S. Davidson, "Information Content of CPU Memory Refencing Behavior," *Fourth Annual Symposium on Computer Architecture*, pp 184-192, March 1977.

[Harr75] Harrison, William, "A Class of Register Allocation Algorithms," RC 5342, IBM Research Report, 1975.

[Hase85] Hasegawa, M. and Y. 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.

[Henn81] Hennessy, J. L., N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI Processor Architecture," *Technical Report No. 223*, Computer Systems Laboratory, Stanford University, Nov., 1981.

[Henn83] Hennessy, J. L., and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Transactions on Programming Languages and Systems* 5, 3, pp. 422-448, July 1983

[Henn84] Hennessy, J. L., "VLSI Processor Architecture," *IEEE Transactions on Computers*, Vol. c-33 No. 12, Dec., 1984.

[Horw66] Horwitz, L.P., R. M. Karp, R. E. Miller, and S. Winograd, "Index Register Allocation," *J. ACM*, pp. 43-61, 13, 1, Jan., 1966.

[HsuP86] Hsu, P. Y., "Highly Concurrent Scalar Processing," Ph. D. Thesis, University of Illinois at Urbana-Champaign, 1986

[Hsu85] Hsu, Wei-Chung, "Register Allocation for VLSI Processors," *UW Computer Science Technical Report #619*, Nov., 1985.

[Hwan84] Hwang, K. and F. A. Briggs, "Computer Architecture and Parallel Processing," McGraw-Hill Book Company, 1984

[Karp85] Karplus, K., and A. Nicolau "Efficient hardware for multi-way jumps and pre-fetches" 18^{th} Annual Microprogramming Workshop, Dec. 1985.

[Kenn72] 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.

[Kogg81] Kogge, P. M., "The Architecture of Pipelined Computers," McGraw-Hill, New York, 1981

[Kran86] Kranz, David, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, "ORBIT: An Optimizing Compiler for Scheme," *SIGPLAN 86' Symposium on Compiler Construction* June, 1986

[Ledg81] Ledgard, H., *Ada: An Introduction*, Springer Verlag, 1981.

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

[Leve81] Leverett, B. W., "Register Allocation in Optimizing Compilers," *Ph.D. Thesis*, CMU CS-81-103, Carnegie-Mellon University, Feb., 1981.

[Liou85] Liou, Koujuch, "Design of Pipelined Memory Systems for Decoupled Architectures" Ph. D. thesis, Computer Science Department, University of Wisconsin-Madison, Oct. 1985.

[Logo81] Logothetis, G., and P. Mishra, "Compiling Short-circuit Boolean Expressions in One Pass," *Software--Practice and Experience,* 11 1197-1241, 1981.

[Lucc67] Luccio, F., "A Comment on Index Register Allocation," CACM, Vol. 10, Number 9, 1967, pp. 572-574.

[Macl84] MacLaren, M. D. "Inline Routines in VAXELN Pascal" *ACM SIGPLAN'84 Symposium on Compiler Construction*, SIGPLAN Notice Vol. 19, No. 6, June, 1984.

[Madh82] Madhavji, N. H. and I. R. Wilson, "CRAY Pascal," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, 1982

[Mccr84] McCreight, E., "The DRAGON Computer System: An Early Overview," NATO Advanced Study Institute on Microarchitecture of VLSI Computers, Urbino, Italy, July, 1984.

[McMa72] McMahon, F. H, "FORTRAN CPU Performance Analysis", Lawrence Livermore Laboratories, 1972

[Milu86] Milutinovic, Veliko, "GaAs Microprocessor Technology" *Computer*, Vol. 19, No. 10, Oct. 1986

[Mosh85] Moshier, M. and V. Rajlich, "Slumload: A Register Allocation Algorithm and its Statistical Analysis," CRL-TR-2-85, University of Michican.

[Moto82] Motorola, *The MC68020 Enhanced M68000 Microprocessor*, Product Review, Motorola Semiconductors, Austin, Texas, March, 1982.

[Myer81] Myers, Glenford J., *Advances in Computer Architecture*, second edition, Intel Corporation, Santa Clara, California, 1981.

[Neff86] Neff, Laura "CLIPPER Microprocessor Architecture Overview," *IEEE Spring Compcon Conference*, 1986.

[Nico85] Nicolau, Alexandru, "Parallelism, Memory Anti-Aliasing and Correctness for Trace-Scheduling" Compilers" *Ph.D. Dissertation*, Yale University, March, 1985.

[Patt80] Patterson, D. A., and C. H. Sequin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Transactions on Computers*, Vol. C-29, No. 2, Feb., 1980.

[Patt82] Patterson, D. A., and C. H. Sequin, "A VLSI RISC," *IEEE Computer*, 15, 9, pp.8-21, Sep., 1982.

[Patt85] Patterson, D. A, "Reduced Instruction Set Computers," *CACM*, Jan., 1985.

[Pfis85] Pfister, G. F., et al., *The IBM Research Parallel Processor Prototype (RP3): Introduction* and Architecture," "Proceedings of the 1985 International Conference on Parallel Processing" Auguest 1985.

[Ples83] A. R. Pleszkun and E. S. Davidson, "A Structured Memory Access Architecture," *1983 International Conference on Parallel Processing*, Bellaire, MI, pp. 461-471, Aug., 1983.

[Pohm83] Pohm, A. V., and O. P. Agrawal, *High-Speed Memory Systems*, Reston, 1983

[Pope77] Popek, J., Horning, J., Lampson, B. *et al*, "Notes on the design of Euclid," *Sigplan Notices*, Vol. 12, No 3, March, 1977.

[Powe84] Powell, Michael L., "A portable optimizing compiler for Modula-2," *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, 1984.

[Radi82] Radin, G., "The 801 Minicomputer," *Symp. on Architecture Support for Programming Languages and Operating Systems*, pp. 39-47, March, 1982.

[Raga83] Ragan-Kelly, R., "Performance of the Pyramid Computer," *Proc. COMPCON*, Feb., 1983.

[Rama77] Ramamoorthy, C. V., and H. Li, "Pipeline Architecture," *Computing Surveys* vol. 9, no. 1, March, pp. 61-102.

[Rose67] Rosen, S., *Programming System and Languages*, McGraw-Hill, N.Y., 1967.

[Russ78] Russell, R. M., "The CRAY-1 Computer System," *Comm. ACM*, pp. 63-72, Jan., 1978.

[Rust72] Rustin, R., *Design and Optimization of Compilers*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Sche77] Scheifler, R. W., "An Analysis of Inline Substitution for a Structured Programming Language," *Comm. ACM*, 20, 9, pp. 647-654, Sep., 1977.

[Seth70] Sethi, R. and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *Joural of the ACM* 17, 6, 1970, pp. 715-728

[Sedg83] Sedgewick, R. *Algorithms* Addison-Wesley 1983.

[Site78] Sites, R. L., "Instruction Ordering for the Cray-1 Computer," Tech. Report, 78-CS-023, UC San Diego, July 1978.

[SmitA82] Smith, A. J., "Cache Memories," *ACM Computing Surveys*, 14, 3, pp. 473-530, Sep., 1982.

[SmitA85] Smith, A. J., "Cache Evaluation and the Impact of Workload Choice," *Proc. 12th Annual International Symposium on Computer Architecture*, June, 1985.

[SmitJ81] Smith, J. E., "A Study of Branch Prediction Strategies," *Proc. Eighth Annual International Symposium on Computer Architecture*, pp. 135-142, May, 1981.

[SmitJ83] Smith, J. E., A. R. Pleszkun, R. H. Katz, and J. R. Goodman, "PIPE: A High Performance VLSI Architecture," *Proc. IEEE International Workshop on Computer System Organization*, pp 131-138, March 1983.

[SmitJ85] Smith, J. E., and J. R. Goodman, "Instruction Cache Replacement Policies and Organizations," *IEEE Transactions on Computers*, Vol. C-34, No.3, March, 1985.

[Thor70] Thornton, J. E., "Design of a Computer, The Control Data 6600," Scott, Foresman and Co., Glenview, Ill., 1970.

[Tjad70] Tjaden, G. S. and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers* 19(10):889-895, October 1970

[Toma67] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journalof Research and Development* 1, pp. 25-33, Jan., 1967.

[Wall86] David W. Wall, "Global Register Allocation at Link Time," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* June 1986

[Youn85] Young, H., "Evaluation of a Decoupled Computer Architecture and the Design of A Vector" Extension," Computer Sciences Technical Report #603, July, 1985