The Relative Effects of Optimization on Instruction Architecture Performance

K. J. Cuderman and M. J. Flynn

Technical Report CSL-TR-89-398

October 1989

Supported by IDA under contract 0031-68 and by NASA under contract NAG2-248.

The Relative Effects of Optimization on Instruction Architecture Performance

by
K. J. Cuderman and M. J. Flynn

Technical Report CSL-TR-89-398
October 1989

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

The Stanford Architect's Workbench is a simulation platform used to evaluate the impact of optimization on the relative performance of instruction set architectures. The total impact optimization makes on an application is the combined interaction of the optimizer, the architecture, and the cache configuration. The relative performance of seven architectures are compared using a suite of six application programs.

Optimization reduces the number of executed instructions, but its effectiveness varies with architecture. Register architectures capitalize on temporaries introduced by optimization without incurring penalties for moving data. Short instructions for register operations reduce the instruction bandwidth in addition to reducing the number of instructions.

Reducing the number of executed instructions does not yield a reduction in memory traffic. Optimization only slightly alters the program working set size. An instruction cache quickly masks the effect of optimization. The result is that the instruction memory traffic remains almost constant for an application.

Key Words and Phrases: Global Optimization, optimization, architecture, instruction bandwidth, instruction traffic, memory traffic, performance, instruction set architecture, registers, register allocation.

Contents

1 Introduction 1

1.1 Architectures Examined 1

2 Optimization Effects on Instruction Execution 2

2.1 Local Optimization 2

2.2 Global Optimization 3

3 Global Optimization Effects on Instruction Bandwidth 7

4 Optimization Effects in the Presence of an Instruction Cache 7

4.1 Execution Time 8

4.2 Relative Traffic Comparisons 9

5 Summary and Future Work 11

A Appendix: Benchmark Characteristics 14

A.1 Benchmark Descriptions 14

A.2 Benchmark Statistics 14

List of Figures

1 Register machine code: for i = 1 to MAX A[i]=B[i]. 5

2 Stack machine code: for i = 1 to MAX A[i]=B[i]. 6

3 The cycles required to execute 100 FLEX instructions. 8

4 Comparing relative instruction traffic with and without global optimization (Normalized to FLEX without optimization). 10

5 Comparing relative instruction traffic with and without global optimization for window architectures (Normalized to FLEX without optimization). 11

6 Relative instruction traffic for various architectures (Normalized to FLEX). 12

List of Tables

1 Average Static and Dynamic size (FLEX only). 1

2 Percent reduction in executed instructions for each benchmark and architecture. 3

3 Effective number of registers for each register based architecture. 4

4 Percentage Increase in the Number of Distinct Instruction Bytes Referenced. 7

5 Percentage Decrease in Required Instruction Bandwidth. 7

6 Benchmarks Static and Dynamic Statistics 14

Optimization	Average Static Bytes	Average Dynamic Instructions	Average Dynamic Bytes	Average Static Bytes Used	Weighted Dynamic Instruction Count (Percent Improvement)
None	29,766	2584,473	9,2 17,000	18,712	
Local	30,046	2555,653	9,3 17,000	18,897	1%
Global	30,334	2,149,139	7,140,000	19,142	17%

Table 1: Average Static and Dynamic size (FLEX only).

1 Introduction

There are two basic reasons for examining the relative effect of optimization on different architectures. The first is to predict the effect of optimization on a given program. The second is to determine if certain optimizations are more effective on particular architectures. These two are related; if optimizations are more effective on certain architectures, then the performance relationships for various architectures will differ when the applications are optimized.

These issues were studied using the Stanford Architect's Workbench facilities [3]. Optimization is performed at the intermediate code level using a U-code to U-code optimizer (UOPT) [4]. The optimizer allows local optimization, local plus global optimization, or optimization plus global many-few register allocation. Since not all of the architectures under consideration are register machines, global many-few register allocation is not used for their evaluation.

The register allocation policy for all the register based architectures is perfect global one-one. Variables are assigned to registers on a procedure basis, and are bound to that register for the whole procedure. The variables are chosen based on the number of uses they had in an actual execution with the same data set, and thus it is the perfect assignment for global one-one allocation [5].

Seven architecture configurations were evaluated using intermediate code with no special optimization, with local optimization, and with global optimization. Six Pascal programs (benchmarks) were analyzed and all data presented is a composite of all six unless explicitly noted otherwise. (See appendix A for benchmarks details.)

1.1 Architectures Examined

DCA The Direct Correspondence Architecture (DCA) has a high degree of correspondence between the architecture and the high level language. There is a direct mapping between high level data types, operations, and variables in the source program to the executable version of the program [2] [8]. **For** this DCA the formats and opcode occupy 6-bits, local branches are encoded in 9 bits, and variables are encoded with 6,12,18... bits depending on the number of variables in the contour (procedure scope) address space.

FIX32 The FIX32 architecture is a generic RISC machine. It is a load-store machine with three-register operations. All instructions take 32 bits. The 32 registers are partitioned as follows: three registers are used as evaluation stack temporaries, 23 are designated for local variables, and the other 6 are used for system maintenance (stack pointer, global pointer, etc.)

FIX32 with windows The FIX32W is the FIX32 architecture with windows (similar to SPARC). It

corresponds to an architecture with four sets of register windows, with each window consisting of eight overlapped registers and eight local registers. The register window model is idealized in that no overflow or underflow overhead is assumed and only the local variables are allocated to registers.

FLEX The FLEX architecture is a FIX32 architecture with some IBM S/370 instructions. The basic register-op-register instruction takes just 16-bits and takes two registers (one source and one source/destination register). One source operand may be in memory (like the IBM S/370 RX instruction format); the memory address is calculated from a register and displacement. This instruction type takes 32 bits. Branch instructions and operations with small immediates also require 32 bits. The 16 registers are partitioned as follows: two registers are used for temporaries, 8 are designated for local variables, and the other 6 are used for system maintenance (stack pointer, global pointer, etc.)

FLEX with windows The FLEXW is the FLEX architecture with windows. It corresponds to an architecture with four sets of register windows, with each window consisting of two overlapped registers and eight local registers. The register window model is idealized in that no overflow or underflow overhead is assumed and only the local variables are allocated to registers.

M68-16 The M68-16 architecture is similar to that of the Motorola 68000 family. It has a 16-bit register-op-register operation, short constants (O-7) and branches are encoded in the instruction, longer constants are sized according to their type. This architecture has 16 general purpose registers like the FLEX architecture.

STACK The STACK architecture is a generic stack machine almost identical to the Pcode stack architecture. All instructions are 32 bits long.

2 Optimization Effects on Instruction Execution

2.1 Local Optimization

The local optimizations done by UOPT. (the U-code to U-code optimizer) are local copy propagation, stack height reduction, and constant arithmetic. None of these should result in a significant instruction traffic reduction. Stack height reduction reduces the number of temporaries required at any one point in time. It does not reduce the number of actual operations performed, and may increase the number of data moves. Local copy propagation can reduce memory loads by introducing temporaries. This does not necessarily reduce the number of instructions. Also since local optimization is performed at the basic block level, the scope limits the likelihood of a significant savings from propagation [1].

Table 1 shows the overall effect of using the optimizer on the instruction stream for the FLEX architecture. Local optimization results in a slightly larger dynamic as well as static size. All the averages except the ***Weighted Dynamic Instruction Count*** are derived from raw data, and thus are the combined unweighted activity of all programs. These numbers are dominated by ***the*** larger programs. ***The Weighted Dynamic Instruction Count*** weighs each program equally.

Local optimization is primarily designed to reduce the number of numeric operations, to perform compile time arithmetic, and to shorten the lifetime of register variables. These optimizations reduce the amount of data traffic to memory, reduce the number of long execution cycles, and reduce the number of registers required at any one time (useful when global many-few allocation is performed).

Benchmark	FIX32	FIX32W	FLEX	FLEXW	STACK	DCA3-6	M68-16
ccal	6	13	4	7	2	6	6
compare	48	48	36	36	5	20	35
kaldel	32	33	32	34	14	26	34
macro	4	8	2	5	2	3	2
pasm	5	6	4	5	-2	3	5
pcomp	5	8	1	3	1	2	2
Average	16.6%	19.3%	13.2%	15.0%	3.7%	10.0%	14.0%

Table 2: Percent reduction in executed instructions for each benchmark and architecture.

By itself local optimization does not have a significant impact on program performance. On the larger programs, the dynamic instruction traffic rose slightly (1 to 2 percent); on the smaller ones it dropped several percent. Although table 1 shows the results for just one architecture the local optimization results apply to the other architectures too. Local optimization had little effect on the number of dynamic instructions executed.

The data traffic showed a reduction in memory reads and writes and increased register usage. Thus a one-one register allocator is able to capitalize on the temporaries created during local optimization. But even so, the reduction in data traffic was also only a few percent. Thus the general conclusion is that local optimization without global optimization has limited effectiveness. The AWB tools do not support global optimization without local optimization, so an evaluation of the effect local optimization plays in global optimization was not studied.

2.2 Global Optimization

Global optimization is performed on a procedure basis rather than by basic block. It creates control flow graphs to detect loops and unreachable code, and performs a full data flow analysis. In UOPT, the data flow analysis phase, in addition to computing flow information, calculates the required transformations [4]. The major optimizations that are performed are copy and constant propagation, global common subexpression elimination, loop-invariant expression removal and partial redundancy elimination, redundant store elimination, and induction variable elimination with the associated strength reduction [1].

2.2.1 Global Optimization Effects on Dynamic Instruction Execution

Performing global optimization (and local optimization) results in fewer instructions being executed for the benchmark suite on each architecture (Table 2). For register based architectures, the percent improvement varies monotonically with the number of registers. Table 3 highlights the register architectures. The FLEX architecture and the M68-16 both have 16 registers, but the M68-16 also allows small constants to be encoded in a register field. Thus all small constants appear as though they were allocated to a register. The FLEXW and FIX32 also have basically the same number of registers, but the multiple windows on FLEXW limit the number of registers that can be used at one time. This means that there are usually less than 32 usable registers at a given time.

Global optimization reduces the number of executed instructions for all architectures, but it is less effective for non-register based machines. The DCA3-6 and the STACK are the only such machines in this study. The direct correspondence architecture (DCA3-6) is much closer to a register machine than the STACK;

Architecture	FLEX	M68-16	FLEXW	FIX32	FIX32W
Registers	16 reg.	16 reg. + small const.	32 constrained in windows	32 reg.	96 constrained in windows
Percent Reduction (Instruction Count)	13.2%	14.0%	15.0%	16.6%	19.3%

Table 3: Effective number of registers for each register based architecture.

it has three operand operations, but operands are in a data buffer. The STACK architecture is a true stack machine where all operand must be loaded on the stack; there are no directly accessible local operands besides top of stack and second form top of stack (in a binary operation). The next section illustrates how the lack of accessible locals results in more data movement and hampers optimization.

2.2.2 Sample Loop Optimization

The benchmark ***compare*** shows the greatest improvement for all the register machines. It has one inner loop that is a single basic block and 72% of all executed instructions are in that block (for FLEX with no optimization). The loop size is halved by doing induction variable elimination, and by moving loop invariant code out of the loop, resulting in a 36% reduction in the total number of executed instructions. For the FLEX architecture, all other optimizations amounted to about 0.25%.

Figure 1 and figure 2 are an example of how a simplified loop is optimized. This example shows the global optimizations performed on a simple copying loop. The loop is the following

- for i = 1 to MAX A[i]=B[i];

Induction variable elimination and code motion (moving static code to the header of the loop) are performed. These figures illustrate why optimization of loops works much better for register based machines than for stack machines.

One basic optimization involves the introduction of temporaries. Induction variable elimination uses address pointer temporaries for each array being accessed. For both stack and register machines, temporaries reduce the amount of address calculations required each time the array is accessed. Variables can remain in registers without any direct overhead. In a stack machine they must be saved and restored multiple times, or at least pushed and manipulated on the stack. This results in more data movement which offsets much of the benefit derived from the reduced address calculations.

2.2.3 Global Optimization Effects on Static Instruction Execution

A program has a physical object code size and a static execution size. The object code size includes the whole application, many parts of which are never executed under norrnal circumstances. The static execution size is the the number of distinct instruction bytes referenced. Thus the static execution size is the instruction traffic a memory system would see with a perfect (or infinite) cache. Table 4 shows the percent change in the number of distinct bytes referenced when optimization is performed.

Comparing the average results in table 2 with those in table 4, there seems to be little correlation between the effect optimization has on static execution size and the effect it has on dynamic instruction count. Some architectures showed an increase in the number of static bytes referenced, while others had a

	Register		Optimized Register	Comments
	Ri ← #1		Ri ← #1	Load i
			Rt1← A	Calculate LHS addr.
			Rt2←Ri-#1	
			Rt2←Rt2*4	
			Rla←Rt1+Rt2	Rla = left address (A[1])
			Rt1←B	Calculate RHS addr
			Rt2←Ri-#1	
			Rt2←Rt2*4	
			Rra←Rt1+Rt2	Rra= right address (B[1])
L1:	Rtl ← B	L1:		Load RHS address
	Rt2←Ri-#1			
	Rt2← Rt2*4			
	Rt2←Rt2+Rt1			
	Rt1← (Rt2)		Rtl ←(Rra)	Load B[i] data
	Rt2←A			Load LHS address
	Rt3←Ri-#1			
	Rt3←Rt3*4			
	Rt3←Rt2+Rt3			
	(Rt3)←Rt1		(Rla)←Rt1	Store data in A[i]
	Ri←Ri+#1			Increment i
			Rra←Rra +#4	Increment Rla and Rra
			Rla←Rla +#4	
	Rt1← #MAX		Rt1← #(MAX*4 + A)	Load Loop limit
	COMPARE Ri ,Rt 1		COMPARE Rla,Rt 1	Do compare
	JUMPLE L1		JMPLE L1	Repeat if LE.

Figure 1: Register machine code: for i = 1 to MAX A[i]=B[i].

	STACK		Optimized STACK	Comments
	LDC #1		LDC #1	Load i
	STORE i		STORE i	
			LOAD A	Calculate
			LOAD i	LHS
			DEC	Base Address
			LDC #4	
			MUL	
			ADD	
			STORE T1	T1 = address A[1]
			LOAD B	Calculate
			LOAD i	RHS
			DEC	Base Address
			LDC #4	
			MUL	
			ADD	
			STORE T2	T2 = address B[1]
L1:	LOAD B	L1:	LOAD T2	Load RHS address
	LOAD i			
	DEC			
	LDC #4			
	MUL			
	ADD			
	ILOAD		ILOAD	Load B[i] data
	LOAD A		LOAD T1	Load LHS address
	LOAD i			
	DEC			
	LDC #4			
	MUL			
	ADD			
	ISTORE		ISTORE	Store data in A[i]
	LOAD i			Increment i
	INC			
	STORE i			
	LOAD i			
			LOAD T1	Increment T1 and T2
			LDC #4	
			ADD	
			STORE T1	
			LOAD T2	
			LDC #4	
			ADD	
			STORE T2	
			LOAD T1	
	LDC #MAX		LDC #(MAX*4 + A)	Load Loop limit
	COMPARE		COMPARE	Do compare
	JUMPLE L1		JMPLE L1	Repeat if LE.

Figure 2: Stack machine code: for i = 1 to MAX A[i]=B[i].

	FIX32	FIX32W	FLEX	FLEXW	STACK	DCA3-6	M68-16
Average Change	2.7%	-0.7%	2.0%	-0.1%	2.0%	-3.0%	0.9%

Table 4: Percentage Increase in the Number of Distinct Instruction Bytes Referenced.

	FIX32	FIX32W	FLEX	FLEXW	STACK	DCA3-6	M68-16
Bandwidth Decrease	-1.0%	-1.3%	4.6%	3.8%	-0.1%	2.7%	10.9%

Table 5: Percentage Decrease in Required Instruction Bandwidth.

decrease. Whether the static execution size increases or decreases is not related to the reduction in instruction stream size.

For the DCA3-6 architecture, optimization reduced the executed code size by 3 percent (the largest reduction of any examined architecture), but had one of the smallest improvements in the number of executed instructions. The FIX32W architecture showed the greatest dynamic improvement and the second greatest improvement in static size.

3 Global Optimization Effects on Instruction Bandwidth

The instruction bandwidth (bytes/instruction) measures the rate the memory system provides instructions relative to the instruction execution rate. Table 5 shows the change in the required bandwidth that occurred from optimization. With optimization, the instruction bandwidth requirement for the denser architectures decreased but it increased for the fixed length instruction architectures.

The change in instruction bandwidth shows that optimization promotes better utilization of registers. The FLEX(W) and the M68-16 architectures all have shorter instructions for operations involving only registers. If registers are used more effectively, the percent of instructions involving only registers increases. Since these are the only 2-byte instructions, this accounts for much of the bandwidth reduction. The bandwidth if the M68-16 is reduced by more than twice that of any other architecture. This is due to the short immediate constant which can also be incorporated in a 2-byte instruction.

The slight increase in bandwidth required for the FIX32(W) architectures is not actually due to optimization, but due to the modeling of the absolute jump instruction for procedure calls. An absolute jump is modeled as a extra long instruction rather than 2 instructions. This is true for all architectures, but it is masked when the overall bandwidth is reduced. The bandwidth increase for the FIX32(W) shows the increased effect of procedure calls. Global optimization does not effect the number of procedure calls, but reduces the instructions executed within them.

4 Optimization Effects in the Presence of an Instruction Cache

The number of instructions executed gives a performance measure for an architecture, but it does not include memory effects. Similarly, the static program size and required instruction bandwidth (to CPU) are independent of memory system configuration. Memory system performance is determined by the

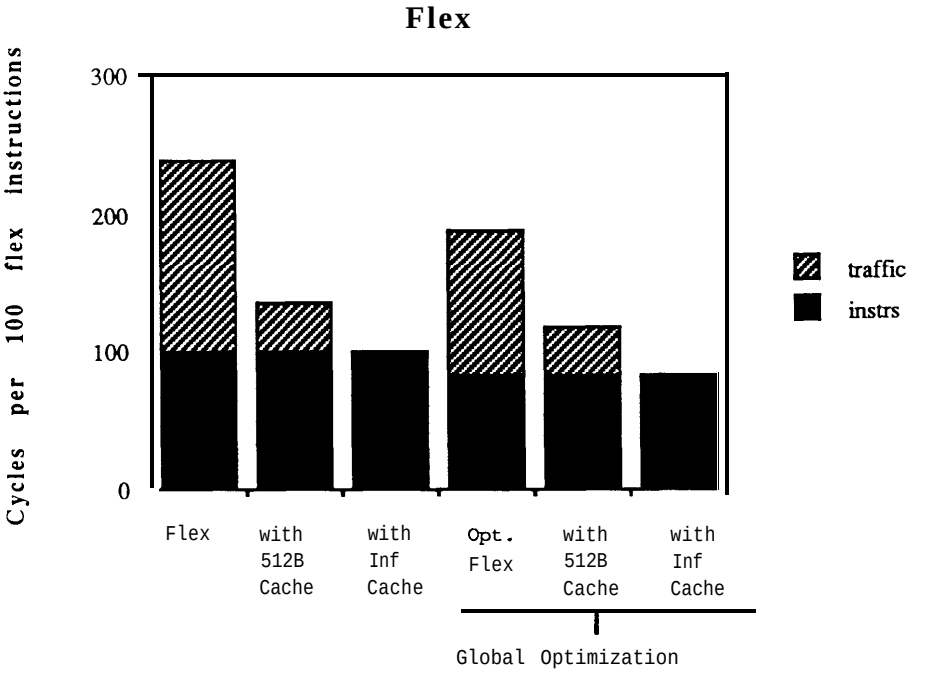

Figure 3: The cycles required to execute 100 FLEX instructions.

number of instructions executed and the program working set characteristics.

4.1 Execution Time

The execution time is a function of the number of instructions, the cache miss cycles and the pipeline stall cycles. The number of memory wait cycles is primarily a function of the architecture and the cache. With a reasonably short pipeline the number of pipeline stalls should not vary much across architectures. Most stalls are due to branches which are generally invariant across architectures and optimization. Optimization directly effects the number of execution cycles and the cache miss cycles. Cache miss cycles are a function of both the cache size and the memory system.

Figure 3 illustrates the effect of both cache and optimization on memory system performance assuming a three cycle cache miss penalty for an 8-byte instruction line. In the case where there is no cache, an 8-byte instruction buffer is filled. The "FLEX" column consists of 100 cycles for instruction execution and the cache miss wait cycles caused by 100 instructions. The next two columns show the effect of a 512 byte cache or a infinite cache on total execution cycles. And the last three columns show the same data after optimization.

Without a cache, each instruction eliminated by optimization saves about two memory wait cycles. The percent reduction in the traffic cycles is **twice** that of the instruction cycles, amplifying the effect of optimization. There was a 17 % reduction in the average number of instructions executed, but the overall cycle savings was 21 %.

The optimization effect is not altered by the presence of a cache. With optimization the number of traffic cycles differs by less than one percent even though there are still a considerable number of wait cycles. For intermediate cache sizes the effect of optimization on the total cycle count is less than the effect on just the number of executed instructions. With the 512 byte cache the overall cycle savings (optimization

plus cache) was only 12.5 %.

With an infinite cache, less than one percent of all cycles would be due to memory waits whether optimization was performed or not. The ratio of static to dynamic size of the program determines the amount of instruction traffic. The static size of the programs determine the amount of instruction traffic for an infinite cache. Thus if the ratio of static to dynamic size is large there may still be a significant amount of memory traffic even with a very large cache. Once the traffic wait cycles become negligible in comparison to the instruction cycles then the overall performance improvement is proportional to instruction cycles saved. These results also apply to the other architectures studied.

4.2 Relative Traffic Comparisons

The cache effectiveness has a critical impact on overall performance; thus it is important to look at the effect of an optimized workload on cache performance. Optimizations rely on reducing the size of loops but do not generally effect the overall code size. With denser loops one might expect that working sets could be captured with smaller caches, but generally little improvement was found. This may by an artifact of the workload or the cache sizes that were studied. The smallest cache size was 512 bytes which may capture many inner loops regardless of the optimization level.

Comparing relative traffic with and without global optimization shows the effect of the architecture on cache performance. All curves in figure 4 are relative to the instruction traffic for the FLEX architecture without optimization. By making all the data relative to one architecture and one optimization level, each benchmark weighs equally in the total For each cache size the traffic is related to the instruction traffic of the FLEX architecture (no global optimization) with the same cache size.

The greatest relative traffic improvements due to optimization are in the absence of a cache, but some configurations have a greater relative traffic degradation. The FIX32, STACK, and FLEX architectures with any size cache, produce at least as many references after the applications are optimized. The STACK architecture with a 2K cache performs five times more references than the FLEX architecture. The rate at which working sets are captured is not uniform across architectures. With optimization, the STACK architecture had six times as much traffic as the FLEX architecture. This 20 % increase in memory traffic shows an increase in working set size.

Optimization reduces the size of loops, but since much of the loop computation is moved into the loop header, the procedure level working set size does not change much. Most leaf procedures are smaller than 512 bytes, with even smaller the principle loops. All the caches are large enough to hold the optimized loops. Thus if an architecture and cache pair do not capture a procedure working set before optimization, they will not capture it after optimization either.

Generally, the effect of global optimization on the instruction traffic is minimal. For small caches and non-windowed architectures where the traffic is still dominated by the dynamic program size, optimization may reduce the traffic. For medium to large caches where the traffic rates are low, optimization has less than a ten percent effect on the traffic and a much lower effect on the overall performance.

Architectures with register windows (FLEXW and FIX32W) had somewhat different relative traffic characteristics. Figure 5 shows the two architectures and their relative traffic. Here optimization improved the working set size of procedures. For any cache size 2K or greater, the traffic after optimization was considerably reduced, improving cache performance for the same size cache. It does not reduce the traffic enough to allow optimized applications to achieve similar performance with a smaller cache.

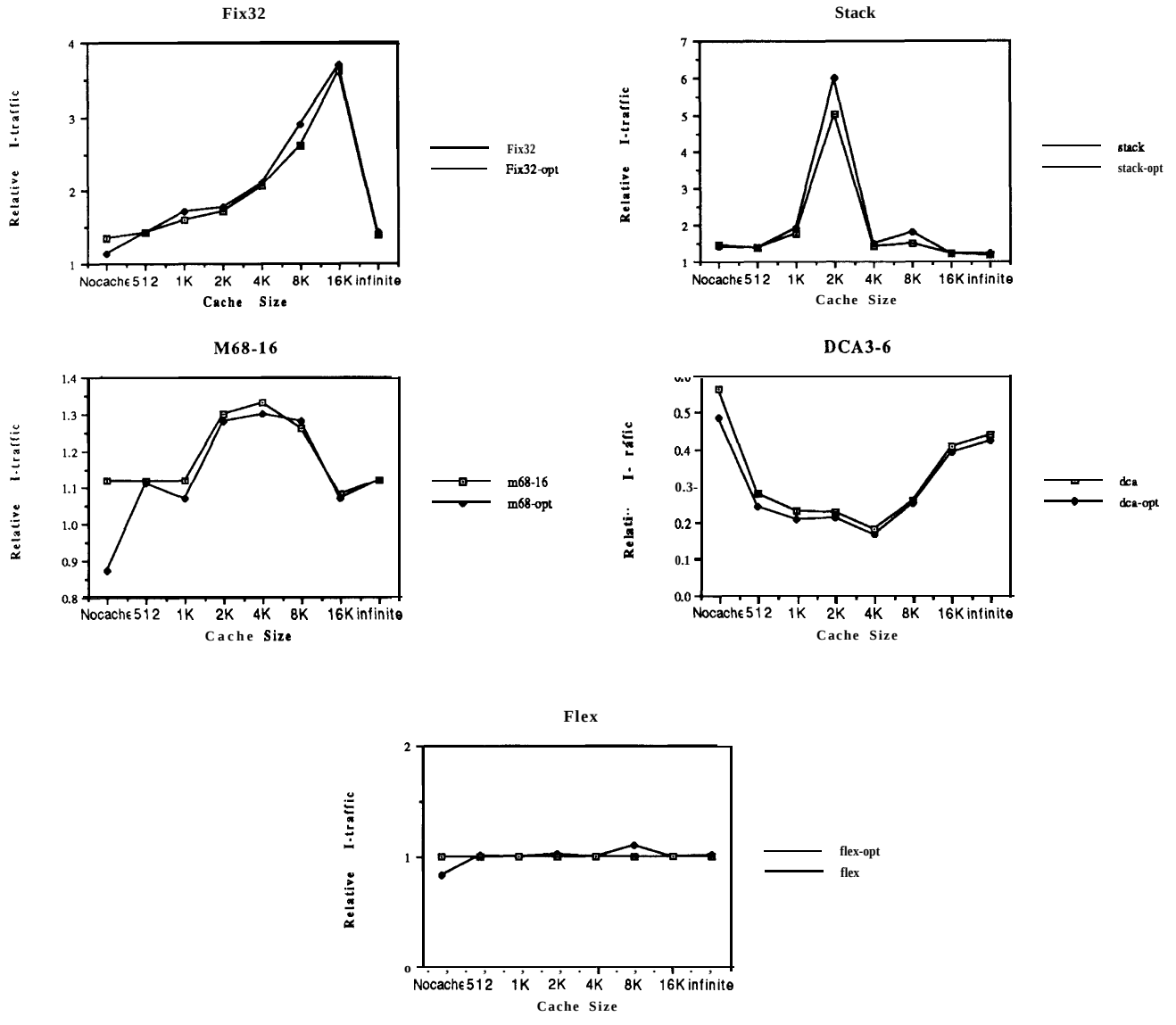

Figure 4: Comparing relative instruction traffic with and without global optimization (Normalized to FLEX without optimization).

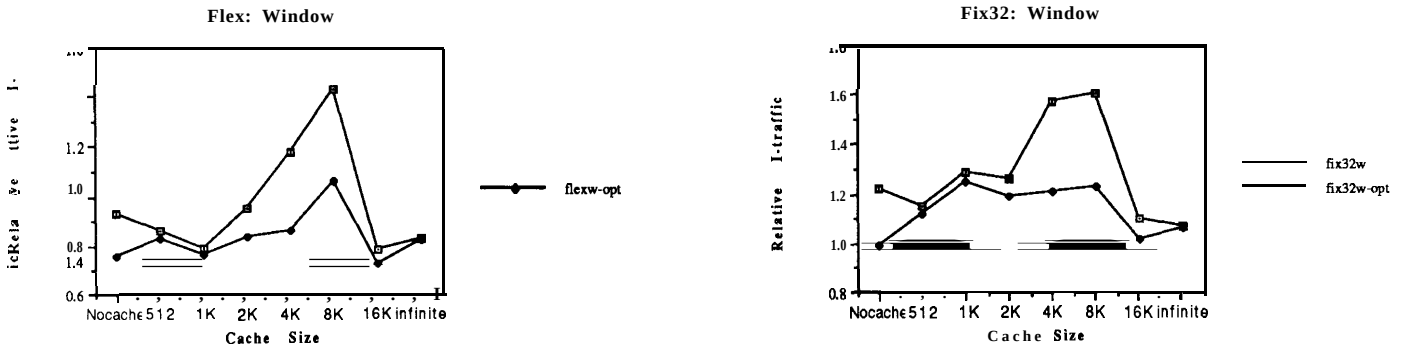

Figure 5: Comparing relative instruction traffic with and without global optimization for window architectures (Normalized to FLEX without optimization).

Comparing optimized with non-optimized workload behavior shows how optimization alters the behavior of the workload. To understand how optimization impacts the relative merits of an architecture and cache choice, the relationship between architectures needs to be examined with and without optimization (Figure 6). Optimization is at most a secondary issue in evaluating the relative traffic created by an architecture. The relative performance of densely encoded architectures is better than less dense architectures independent of optimization.

5 Summary and Future Work

Optimization alters the number of instructions executed, the static program size, the required instruction bandwidth, and the instruction traffic to main memory in different ways. It has a greater impact on certain architectures. Register architectures showed the greatest percent reduction in executed instructions.

The number of available registers seems to be the crucial factor in the effectiveness of optimization. Optimization introduces temporaries which get assigned to registers and eliminate many instructions associated with data loads and stores. Optimization promotes register usage; this results in a fewer bytes being required per instruction on architectures that have shorter instructions for register operations.

The effect of global optimization on the instruction traffic is minimal once the cache is large enough to capture most of the program working set. For small caches there should be considerable reduction in both traffic and the associated cache miss penalties because the traffic is still largely a function of the dynamic program size (which optimization reduces). For medium to large caches where the traffic rates are low, it generally has less than a ten percent effect on the traffic and a much lower effect on the overall performance. If, however, the architecture has register windows, the working set size of procedures is reduced and medium sized caches are much more effective after optimization of the application.

Future work exploring the impact and role of registers and register allocation on the instruction stream

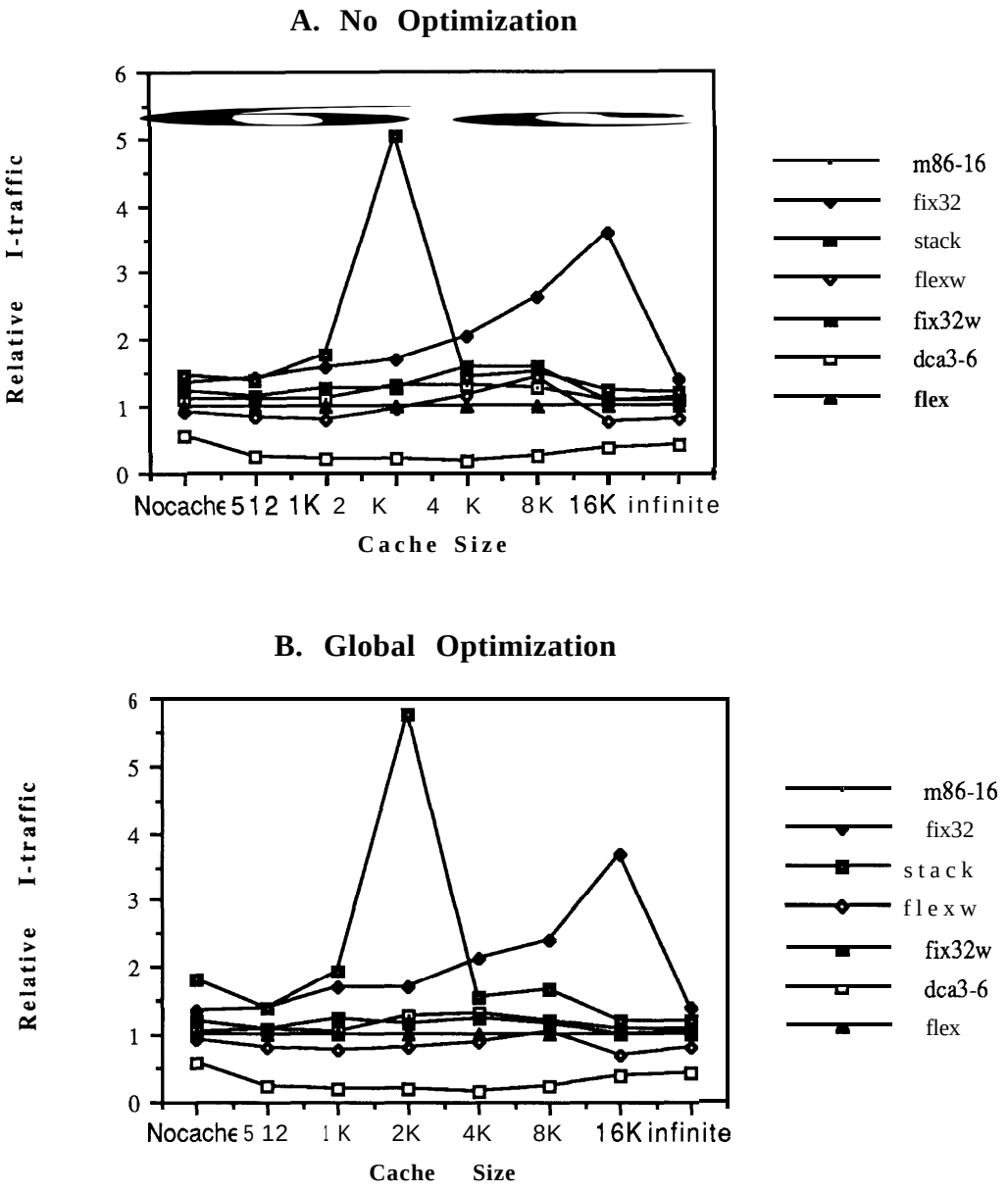

Figure 6: Relative instruction traffic for various architectures (Normalized to FLEX).

needs to be done. Other work [7] studied the effect of data buffering on the data stream. Register allocation techniques significantly reduce the data memory traffic. Since data and instruction reuse and access patterns usually differ the results from the data stream cannot be extended directly to the instruction stream. It is important to understand the interdependence between data buffering and instruction execution. Without such knowledge it is impossible to predict the impact of buffering techniques on the rate of memory requests.

Program	Object Code Size (bytes)	Dynamic Instruction Count	Dynamic Bytes Referenced
ccal	12,160	**949,596**	3,006,630
compare	7,368	6,49 1,032	23,196,556
kaldel	10,208	209,506	636,210
macro	64,336	527,835	1,685,218
pasm	13,168	2,968,147	10,824,628
pcomp	78,384	4,360,722	15,951,368

Table 6: Benchmarks Static and Dynamic Statistics

A Appendix: Benchmark Characteristics

The programs are applications covering a variety of programming styles and functions. All are Pascal programs of medium size. They represent program generation, file processing, calculation and numeric computation. These same programs have been used in earlier studies [2, 6, 7]

A.1 Benchmark Descriptions

- **ccal** is an interactive desk calculator. It reads a script of calculations from a text file and produces results in another text file.
- **compare** is a program which compares two text files and produces a description of their differences (similar to the Unix diff program).
- **kaldel** is a kalman filtering program.
- **macro** is a macro expander for the SCALD computer-aided design system.
- **pasm** is an assembler assembling a **P-code** representation of itself.
- **pcomp** is a Pascal to P-code compiler, which parses its input by recursive descent. The program compiled by pcomp is pasm.

A.2 Benchmark Statistics

Table 6 shows the static and dynamic characteristics for each program. This is data for the FLEX architecture without code optimization and with one-one register allocation.

References

[1] A. Aho, R. Sethi, and J. Ullman. ***Compilers: Principles, Techniques, and Tools.*** Addison-Wesley, 1986.

[2] Donald Alpert. Memory hierarchies for directly executed language microprocessors. Technical Report 84-260, Computer Systems Laboratory, Stanford University, Stanford, California 94305, June 1984.

[3] B. Bray, K. Cuderman, M. Flynn, and A. Zimmerman. The computer architect's workbench. In ***Proceedings of the INFORMATION PROCESSING '89 Conference,*** pages 509-514. IFIP, August 1989.

[4] Frederick ***C.*** Chow. ***A Portable Machine-Independent Global Optimizer-Design and Measurements.*** PhD thesis, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, December 1983.

[5] **R. A.** Freiburghouse. Register allocation via usage counts. ***Communications of the ACM,*** 17(11):638–642, November 1974.

[6] Chad L. Mitchell. ***Processor Architecture and Cache*** *Performance.* PhD thesis, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, July 1986.

[7] Johannes M. Mulder. ***Tradeoffs in Processor-Architecture and Data-Buffer Design.*** PhD thesis, Stanford University, January 1988.

[S] Scott Wakefield. Studies in execution architectures. Technical Report No. 237, Computer Systems Laboratory, Stanford University, January 1983.