Achieving High Instruction Cache Performance with an Optimizing Compiler

Wen-mei W. Hwu and Pohua P. Chang

Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801
(217) 244-8270
hwu@bach.csg.uiuc.edu

Abstract

Increasing the execution power requires a high instruction issue bandwidth, and decreasing instruction encoding and applying some code improving techniques cause code expansion. Therefore, the instruction memory hierarchy performance has become an important factor of the system performance. An instruction placement algorithm has been implemented in the IMPACT-I (Illinois Microarchitecture Project using Advanced Compiler Technology - Stage I) C compiler to maximize the sequential and spatial localities, and to minimize mapping conflicts. This approach achieves low cache miss ratios and low memory traffic ratios for small, fast instruction caches with little hardware overhead. For ten realistic UNIX* programs, we report low miss ratios (average 0.5%) and low memory traffic ratios (average 8%) for a 2048-byte, direct-mapped instruction cache using 64-byte blocks. This result compares favorably with the fully associative cache results reported by other researchers. We also present the effect of cache size, block size, block sectoring, and partial loading on the cache performance. The code performance with instruction placement optimization is shown to be stable across architectures with different instruction encoding density.

Keywords : Cache, Memory hierarchy, Compiler optimization, Instruction placement, Data locality

* UNIX is a Trademark of Bell Laboratories

This research has been supported by the National Science Fundation (NSF) under Grant MIP-8809478, a donation from NCR, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS), the Office of Naval Research under Contract N00014-88-K-0656, and the University of Illinois Campus Research Board.

1. Introduction

1.1. Motivation

The instruction memory hierarchy has received only moderate attention because conventional machines typically have high microcycle count per instruction, and thus, demand low instruction bandwidth. For instance, it takes a VAX-11/780 10.5 microcycles to execute every 3.8 bytes of instructions[1]. An 8-byte instruction buffer that prefetches instructions during idle cache cycles provides enough instruction bandwidth for the VAX-11/780 microengine. In response to the increasing demand for processor speed, performance improving techniques such as pipelining have been widely used to implement processors that require much higher instruction bandwidth. For example, the VAX 8600 implementation requires 3.8 instruction bytes for every 6 microcycles. Futher reducing the number of microcycles per instruction increases the instruction memory bandwidth requirement and suggests the need for better instruction memory hierarchy designs.

Many processor architectures have adopted instruction formats and semantics which allow the instruction units to be efficiently pipelined[2-5]. To simplify instruction decoding, these processor architectures specify fixed instruction formats that prevent conventional encoding techniques from reducing program size. To simplify instruction sequencing, these processors prevent the use of powerful opcodes to encode sequences of microinstructions. Therefore, the instruction unit pipeline becomes more efficient and matches the speed of the execution pipeline. The cost is an increase in the dynamic code size and consequently an increase of the instruction bandwidth requirement.

Compiler code improvement techniques often increase code size. Inline expansion reduces function call overhead[6]. Loop unrolling increases code scheduling flexibility[7]. Trace scheduling extracts the program parallelism[7-8]. These techniques all increase code size and rely on the instruction memory hierarchy to absorb the code expansion cost. This adds further demand on the instruction memory hierarchy performance.

One conventional approach to improving the memory hierarchy performance is to increase the size and/or set-associativity of the top level cache memory[9-10]. For example, the MIPS-X processor uses an 2048-byte, 8-way set-associative instruction cache. This approach is limited by the fact that the cache cycle time increases as the size and set-associativity increase and the fact that only a limited amount of hardware is available[11-14]. To make it worse, if the compiler generates code with little spatial locality and/or many cache mapping conflicts, no cache of reasonable size and set-associativity can provide enough instruction bandwidth.

With advances in the compiler technology, an increasing number of microarchitecture design parameters have been exposed to the compiler. Examples of this trend include pipeline latency[15-16], parallel data paths[7-8,17], and register-file structure[18-20]. The advantage of exposing these microarchitecture details to the compiler is that the compiler can generate code to utilize these microarchitecture features without expensive hardware interlocking schemes. We believe that the instruction memory hierarchy should also be exposed to the compiler for improving the system performance.

1.2. Our Approach

In this paper, we present an instruction placement algorithm which improves the efficiency of caching in the instruction memory hierarchy. Based on dynamic profiling, this algorithm maximizes the sequential and spatial localities, and minimizes cache mapping conflicts of the instruction accesses. The instruction placement algorithm has been implemented in the IMPACT-I C Compiler and produced instruction placement for realistic C programs. The instruction placement for each program is based on the execution of millions of instructions using typical input files.

The instruction cache performance for each program, after applying the instruction placement algorithm, is measured by trace driven simulation. We demonstrate that the instruction layout algorithm can efficiently exploit small (about 2048B), direct-mapped instruction caches with large (about 64B) blocks. Direct mapped caches with large blocks are desirable due to their low control overhead (tag store and hit detection logic). The effect of varying the cache design parameters (cache size, block size, block sectoring, partial loading) is presented. We also demonstrate that direct mapped caches with instruction placement optimization compare favorable with fully associative caches without instruction placement optimization, and perform equally well for different code densities.

1.3. Organization of the Paper

This paper is organized into five sections. Section two describes related work in program optimization for memory system performance. Section three describes the IMPACT-I instruction placement algorithm. Section four presents measurement results of the instruction memory hierarchy performance using the IMPACT-I instruction layout algorithm. Section five concludes the paper and outlines the present research issues in the management and performance measurement of the instruction memory hierarchy. The Appendix gives an outline of the IMPACT-I instrcution placement algorithm.

2. Background

2.1. Previous Work

The problem of restructuring programs for memory system performance has been studied by various researchers. Ferrari examined the potential of restructuring programs to improve program paging behavior[21]. Data alignment methods based on data dependence analysis for highly iterative scientific codes have been observed to improve the performance of cache and local memory organizations[22-24]. Hartley described a function-level program restructuring technique to improve the page-level locality of references and to reduce the number of page faults, using the call graph[25]. On the cache level, Przybylski showed that set associativity increases overall execution time if it increases the cycle time by more than a few nanoseconds[14]. McFarling showed that, by using profile information and excluding certain instructions from the instruction cache, his program restructuring algorithm significantly increased the performance of direct-mapped instruction caches[26].

2.2. Design Target

The goal of the IMPACT-I C Compiler instruction placement mechanism is to layout the target program to maximize the sequential and spatial localities, and to minimize the mapping conflict. To maximize the sequential locality, basic blocks are grouped into a trace if they tend to execute in a sequence. To maximize the spatial locality, instructions are mapped close to each other in the memory space if they are executed close to each other in time. Therefore, almost all the bytes in a memory block will be used when that block is brought in cache. Cache mapping conflicts are minimized by placing the functions with overlapping lifetime into memory locations which do not contend with each other in cache. With compile-time program restructuring, a direct-mapped instruction cache should compare favorably with a fully associative cache.

A.J. Smith has reported realistic values for the miss ratio as a function of cache size and line size[10,27-28]. Table 1 lists a small subset of the design target miss ratios reported by Smith for fully associative instruction cache[28]. Each column gives the expected miss ratios for a fixed block size and varying cache sizes. Each row presents the expected miss ratios for a fixed cache size and varying block sizes. For example, a 2048-byte fully instruction cache with 64-byte blocks is expected to give a 6.8% miss ratio. For another example, a 1024-byte fully associative instruction cache with 32-byte blocks is expected to give a 15.9% miss ratio. We will use the miss ratios in Table 1 as the basis for evaluating the effectiveness of our instruction placement optimization.

To minimize the effect of workload dependent and system dependent factors on the miss ratio, we use typical-size input data to generate traces and the entire execution traces are applied to the cache simulator. For small caches, the effect of context switch is not important. We also conduct a code scaling experiment to show that the instruction cache performance is not sensitive to the increase in the degree of instruction encoding.

3. Instruction Placement

IMPACT-I instruction placement is implemented in five major steps: execution profiling, function inline expansion, trace selection, function layout, and global layout.

Step 1. Execution profiling. In our C compiler, a program is represented by a weigthed call graph. A call graph is a directed graph where every node is a function and every arc is a function call. A weighted call graph is a call graph in which all the nodes and arcs are marked with their execution frequencies.

Each node of the weighted call graph is represented by a weighted control graph. A control graph for a function is a directed graph where every node is a basic block, and every arc is a branch path between two basic blocks. A weighted control graph is a control graph in which all the nodes and arcs are marked with their execution frequencies.

The IMPACT-I profiler translates each target C program into an equivalent C program with additional probe function calls. When the equivalent C program is executed, these probe function calls record the weights of nodes and arcs of the call graph for the entire program and the control graph for each function. It is critical that the inputs used for executing the equivalent C program be representative. Therefore, this approach is very suitable for characterizing realistic programs for which representative inputs can be easily collected. The IMPACT-I Profiler to C Compiler interface allows the profile information to be automatically used by the IMPACT-I C Compiler.

Step 2. Function inline expansion. The function calls (arcs in the weighted call graph) with high execution count are replaced with the function body if possible[6]. The goal is to transform all the important inter-function control transfers into intra-function control transfers. Inline expansion reduces the dynamic inter-function control transfers to a small percentage (about 1%) of all control transfers, which provides two major advantages. First, the spatial locality is increased in that almost all control transfers are within individual functions. Second, removing function calls also reduces potential cache mapping conflicts among functions.

Step 3. Trace* selection. For each function, basic blocks which tend to execute in sequence are grouped into traces. The traces are the basic units of instruction placement to maximize the sequential and spatial localities. A recent paper gave detailed description and evaluation of the trace selection algorithm of the IMPACT-I C Compiler[29]. Note that the inline expansion step provides large functions to enhance the size of the traces selected.

Step 4. Function layout. By carefully placing traces of each function in a sequential order, spatial locality can be further preserved. We start with the function entrance trace, and expand the placement by placing the most important descendent after it. We grow the placement until all the traces with non-zero execution count (profiled count) have all been placed. Traces with zero execution count (profiled count) are moved to the bottom of the function. This results in smaller effective function body, and allows more effective parts of functions to be packed into each page.

Step 5. Global layout. Each function is assumed to have two parts: the effective and non-executed parts. The goal of the global layout algorithm is to place functions which are executed close to each other in time into the same page, so that inter-function cache conflicts are further reduced (already reduced by inline expansion).

To evaluate the effectiveness of our code layout scheme, we randomly select one input for each benchmark to take the traces of dynamic instruction accesses. These dynamic traces include instruction accesses to both the user code and the library code; they do not include any access to the kernel code. In summary, the IMPACT-I instruction placement is based on profile information and the performance evaluation presented in this paper is based on trace driven simulation.

	Block Size (bytes)			
cache size (bytes)	16	32	64	128
512	23.0%	15.9%	11.9%	10.8%
1024	20.0%	13.4%	9.8%	8.4%
2048	15.0%	9.8%	6.8%	5.7%
4096	10.0%	6.3%	4.3%	3.2%

Table 1. *Design Target Miss Ratio (Fully Associative)*

4. Experimentation

Table 2 summarizes several important characteristics of our benchmarks. The *C lines* column shows the static code size of the C benchmark programs measured in the number of program lines. The *runs* column gives the number of different inputs used in the profiling process. The *instructions* column gives the dynamic code size of the benchmark programs, measured in number of dynamic instructions. The *control* column gives the dynamic count of control transfers other than function call/return executed during the profiling process. Both *instructions* and *control* are accumulated among all the runs. The *input description* describes the nature of the inputs used in the profiling process.

4.1. Basic Experiments

4.1.1. Inline Expansion Results

Table 3 offers the inline expansion results. The *code inc* column gives the percentage of increase in static code size due to inline expansion. The *call dec* column gives the percentage of dynamic function calls eliminated by the inline expansion. The *DI's per call* column gives the average number of dynamic instructions executed between dynamic function calls after inline expansion. The *CT's per call* column gives the average number of dynamic control transfers executed between dynamic function calls after inline expansion.

For most of the benchmark programs, the inline expansion mechanism successfully eliminates a large percentage of the dynamic function calls. For programs with infrequent function calls to begin with, the inline expansion mechanism does not eliminate large percentages of dynamic function calls. This is a desirable trait because the overall goal is to ensure infrequent function calls rather than to achieve high elimination percentages.

* The term *trace* here is used as in the *trace scheduling* for global microcode compaction. It is not used as in the *trace driven simulation*. In this paper, if the term *trace* is used as in the *trace driven simulation*, it will appear as *dynamic trace* whenever an ambiguity may occur.

name	C lines	runs	instructions	control	input description
cccp	4660	20	11.7M	2.2M	C programs (100-3000 lines)
cmp	371	16	2.2M	0.5M	similar/disimilar text files
compress	1941	20	19.6M	3.1M	same as cccp
grep	1302	20	47.1M	17.1M	exercised various options
lex	3251	4	3052.6M	1125.9M	lexers for C, Lisp, awk, and pic
make	7043	20	152.6M	32.4M	makefiles for cccp, compress, etc.
tee	1063	18	0.43M	0.17M	text files (100-3000 lines)
tar	3186	14	11M	1.5M	save/extract files
wc	345	20	7.8M	2.2M	same as cccp
yacc	3333	8	313.4M	78.7M	grammar for a C compiler, etc.

Table 2. Profile Results

name	code inc	call dec	DI's per call	CT's per call
cccp	17%	55%	506	95
cmp	3%	49%	265	58
compress	4%	91%	2324	368
grep	31%	99%	11214	4071
lex	23%	77%	7807	2880
make	34%	59%	388	82
tee	0%	0%	15	6
tar	16%	43%	983	127
wc	0%	0%	18310	5146
yacc	24%	80%	1205	303

Table 3. Inline Expansion Results

The program *tee* is a special case where data is copied from the input to the output by system calls (read, write), without much additional computation. Since system calls can not be inline expanded, the call frequency of *tee* is extremely high.

After inline expansion, the frequency of function calls is much smaller than the frequency of intra-function control transitions (branches). It is also observed that hundreds of dynamic instructions are executed per function call. The obvious gain is that register save and restore costs across function boundaries are greatly reduced. A more subtle advantage that directly affects the performance of our instruction placement algorithm is that the function inline expansion mechanism enlarges function bodies and reduces inter-function interactions. More sequential and spatial localities can be found in larger function bodies. Reducing inter-function interactions also removes potential cache mapping conflicts among interacting functions. The result is that most of the complexity in the global layout process can be shifted to the intra-function layout process (trace selection and placement) which is much simpler to implement. We have thus decided to implement a simple global layout process based on a variant of the depth-first-search algorithm (see appendix).

4.1.2. Trace Selection Results

Table 4 presents the trace selection results. The *neutral* column gives the percentage of control transfers from the end of a trace to the start of a trace. The average percentage (about 39%) for this category suggests that a careful selection of a linear ordering of traces could increase the spatial locality significantly. The *undesirable* column gives the percentage of control transfers which enter and/or exit traces at a non-terminal basic block. The *desirable* column gives the percentage of control transfers which go from a basic block to its successor in a trace. The small average percentage (about 3%) in the *undesirable* column and the large average percentage (about 58%) in the *desirable* column indicate that once the control is transferred into a trace, it is likely to remain through the end of that trace. This justifies our approach to use the traces as units of instruction placement. The *trace length* column gives the average number of basic blocks in each trace. On the average, each trace contains 3.4 basic blocks. Since each basic block in the IMPACT-I code contains about 4 machine instructions (4 bytes each), the unit of instruction placements contains about 54 bytes. Considering the spatial locality among traces, a reasonable prediction of a good instruction block size would be about 64 bytes.

We use trace-based analysis to evaluate the effectiveness of our code layout scheme. A trace is generated by feeding a randomly selected input (typical size) to each benchmark program. The entire execution traces are used. These dynamic traces include both user code and library code, but not kernel code. We do not simulate the effect of context swaps.

4.1.3. Memory Access Characteristics

Table 5 shows the instruction memory access characteristics of the benchmark programs and their corresponding dynamic traces. The *total static bytes* column gives the number of machine code bytes generated for each benchmark program. The *effective static bytes* column gives the number of machine code bytes which have non-trivial execution count. The *dynamic accesses* gives the number of dynamic instruction accesses recorded in each dynamic trace.

The effective static program size ranges from 2K to 34K whereas the total static program size ranges from 2.8K to 55K.

name	neutral	undesirable	desirable	trace length
cccp	55.23%	3.74%	41.05%	1.8
cmp	12.74%	4.23%	83.03%	6.9
compress	35.04%	3.15%	61.85%	2.8
grep	20.96%	1.80%	77.24%	4.7
lex	35.02%	1.79%	63.19%	2.8
make	53.93%	2.08%	43.99%	1.8
tar	86.85%	0.38%	12.77%	1.2
tee	24.77%	0.24%	75.00%	4.0
wc	15.09%	9.02%	75.88%	5.5
yacc	49.13%	4.62%	46.25%	2.0

Table 4. Trace Selection Results

name	total static bytes	effective static bytes	dynamic accesses
cccp	51.6K	29.6K	1.5M
cmp	2.8K	2.0K	0.3M
compress	15.6K	8.8K	2.8M
grep	11.1K	4.5K	0.1M
lex	40.4K	29.7K	51.9M
make	55.0K	34.1K	1.8M
tar	25.8K	15.7K	0.2M
tee	6.5K	3.4K	0.1M
wc	3.1K	2.6K	1.1M
yacc	35.7K	27.0K	3.3M

Table 5. Static and Dynamic Code Sizes of Benchmarks

Since the IMPACT-I compiler places the effective and ineffective parts of the program into different pages, only the effective part needs to be accomodated in the main and cache memories. As a result, when a page is transferred from the secondary memory to the main memory, all the bytes of that page are likely to be used.

4.2. Caching Experiments

The primary goal of the IMPACT-I instruction placement mechanism is to improve the cache performance and to reduce the cost of instruction memory hierarchy. For the instruction caches, the goal is to minimize the data storage size and the control overhead (set-associativity and tag storage) to obtain the desired cache hit ratio and memory traffic. Direct-mapped caches are used in all the measurements due to their minimal set-associativity overhead. The next two tables present the effectiveness of the instruction placement mechanism for minimizing the data storage and the tag storage.

4.2.1. Basic Organization

Table 6 shows the effect of varying cache size for a fixed block size (64 bytes). The *miss* columns give the cache miss ratios. The *traffic* columns give the ratios of the number of main memory accesses over the number of dynamic instruction accesses (memory traffic ratio). Note that for the block size of 64 bytes, a 2K-byte instruction cache provides a low miss ratio (average 0.5%) with a reasonable memory traffic ratio (average 8%). As a result, less than 1% of instruction accesses need to wait for the data from an outside cache or the main memory. Also, the bus to the outside cache and the main memory is only loaded by 8% of the instruction access traffic. Even a small instruction cache of 512 bytes provides a reasonable miss ratio (average 1.4%) with a moderate memory traffic ratio (average 22%). Comparing the cache sizes to the static program sizes reveals that the instruction placement algorithm is successful in mapping the programs into small caches.

Table 7 shows the effect of varying the block size for a fixed cache size of 2048 bytes. In general, the miss ratios decrease and the memory traffic ratios increase as the block size increases. The miss ratios decrease with the increase of the block size because each cache miss brings in more useful bytes for larger block sizes. The instruction placement algorithm maximizes this effect by placing the bytes which are accessed close in time in the same block. The traffic ratios increase with the increase of the block size because each cache miss brings in more useless bytes for large block sizes. The instruction placement mechanism minimizes this effect also by placing in the same block the bytes which are accessed close in time.

For a fixed cache size, the larger the block size, the smaller the number of tags that are required to manage the cache. For a 2K-byte instruction cache, the 64-byte block size provides a low miss ratio (average 0.5%) and reasonable memory traffic ratio (average 8%). The configuration requires only 16 tags, successfully minimizing the control overhead. Assuming 4 bytes of tag space for each block, we have a total of 64 bytes of tag space for the entire cache. The overhead is only (64-bytes / 2048-bytes) and is approximately 3% of the data store size.

Note that the memory traffic ratio is rather high for benchmarks *cccp* and *make*. Also, since the cache miss penalty increases with the block size, the effective cache access time may increase inspite of the decreased miss ratio. For some systems (especially multiprocessor systems), it is desirable to decrease the memory traffic ratio and the cache miss penalty at the cost of increasing the miss ratio.

We assume that the memory or secondary cache is interleaved and can deliver one data per cycle after the initial access

name	8K		4K		2K		1K		0.5K	
	miss	traffic	miss	traffic	miss	traffic	miss	traffic	miss	traffic
cccp	0.86%	13.79%	1.53%	24.40%	2.70%	43.13%	3.52%	56.32%	4.24%	67.87%
cmp	0.01%	0.15%	0.01%	0.15%	0.01%	0.15%	0.01%	0.15%	0.01%	0.17%
compress	0.00%	0.07%	0.00%	0.08%	0.01%	0.08%	0.01%	0.09%	3.54%	56.63%
grep	0.06%	0.88%	0.06%	0.91%	0.06%	0.87%	0.07%	1.11%	0.60%	9.62%
lex	0.01%	0.09%	0.01%	0.21%	0.03%	0.48%	0.06%	0.93%	0.31%	4.96%
make	0.32%	5.06%	0.69%	11.10%	1.35%	21.59%	2.03%	32.46%	2.44%	39.02%
tar	0.09%	1.51%	0.24%	3.88%	0.27%	4.27%	0.42%	6.76%	0.61%	9.79%
tee	0.06%	0.92%	0.06%	0.092	0.08%	1.2%	0.08%	1.28%	0.08%	1.33%
wc	0.00%	0.06%	0.00%	0.06%	0.00%	0.06%	0.00%	0.06%	0.00%	0.06%
yacc	0.02%	0.28%	0.23%	3.64%	0.49%	7.86%	1.17%	18.73%	1.99%	31.89%

Table 6. The Effect of Varying the Cache Size

delay. We also assume that the data for which the cache miss occurs is the first data delivered after the initial memory access delay (load forwarding). To furthur reduce the cache miss penalty, the processor resumes execution as soon as the accessed data comes back from main memory (early continuation). Subsequent instruction fetches after a cache miss, if sequential, can directly obtain the instructions from the memory bus as the cache block is being repaired (streaming). For a taken branch before the block is completely filled, the CPU is stalled until the block is completely transferred.

For a 64-byte block size and a 4-byte memory bus, 16 cycles are required after the initial memory access to complete the block transfer. Due to the large transfer size, the CPU may be stalled. Our layout algorithm does not guarantee that the data for which the repair sequence is incurred is positioned at the beginning of the cache block. Therefore, the CPU is stalled while repairing the part of the cache block in front of the data for which the miss is incurred. The average number of stalled cycles caused by each cache miss is about half of the block, assuming random access pattern. For a 64-byte block and a 4-byte memory bus, the CPU is stalled for about 8 cycles.

name	16B		32B		64B		128B	
	miss	traffic	miss	traffic	miss	traffic	miss	traffic
cccp	7.53%	30.10%	4.32%	34.58%	2.70%	43.13%	2.10%	67.33%
cmp	0.04%	0.15%	0.02%	0.15%	0.01%	0.15%	0.01%	0.16%
compress	0.02%	0.07%	0.01%	0.08%	0.01%	0.08%	0.00%	0.09%
grep	0.19%	0.76%	0.10%	0.82%	0.06%	0.91%	0.03%	1.01%
lex	0.08%	0.33%	0.05%	0.38%	0.03%	0.48%	0.02%	0.69%
make	4.24%	16.95%	2.40%	19.19%	1.35%	21.59%	0.95%	30.39%
tar	0.72%	2.90%	0.42%	3.32%	0.27%	4.27%	0.20%	6.37%
tee	0.25%	0.98%	0.13%	1.06%	0.08%	1.20%	0.04%	1.41%
wc	0.01%	0.06%	0.01%	0.06%	0.00%	0.06%	0.00%	0.06%
yacc	1.13%	4.53%	0.66%	5.25%	0.49%	7.86%	0.52%	16.78%

Table 7. The Effect of Varying the Block Size

Including the initial memory access cost, the effective cache access time may increase although the miss ratio is lower than, for example, the 32 byte block size configuration.

4.2.2. Reducing Memory Traffic

One approach to decreasing the memory traffic ratio and the cache miss penalty while increasing the miss ratio is to partition each block into sectors and only bring in the accessed sector upon cache miss. The memory traffic ratio is reduced because the number of memory accesses caused by each cache miss is reduced to the size of each sector (rather than the size of each block), and thus fewer unused words are fetched. The miss ratio increases because the spatial locality is not fully exploited. Since the instructions placed into the same block are likely to be executed near each other in time, not bringing in the rest of a missing block can be expected to cause more cache misses.

The *sector* column in Table 8 presents the effect of dividing the 64B blocks into sectors of 8 bytes each for a 2048B cache. A comparison with the *64B* column in Table 7 shows that, for programs causing large memory traffic ratios, sectoring the blocks decreases the memory traffic ratio at the cost of increasing the miss ratio. The problem with this approach is that it increases the miss ratio to such a degree (e.g. cccp) that the average cache access time can actually increase.

An alternative scheme is to load only part of the missing block, from the accessed location to the end of that block or to a valid entry previously loaded in. The processor resumes execu-

name	sector		partial			
	miss	traffic	miss	traffic	avg.fetch	avg.exec
cccp	13.88%	27.76%	2.86%	33.78%	11.8	8.2
cmp	0.33%	0.65%	0.05%	0.66%	14.2	12.3
compress	0.47%	0.94%	0.07%	0.99%	13.9	12.0
grep	0.11%	0.21%	0.02%	0.24%	12.6	9.9
lex	0.18%	0.35%	0.04%	0.41%	11.1	7.8
make	8.82%	17.64%	1.52%	19.77%	13.0	10.1
tar	1.62%	3.25%	0.28%	3.55%	12.8	12.2
tee	1.31%	2.62%	0.21%	3.00%	14.0	9.9
wc	0.16%	0.33%	0.02%	0.33%	14.9	12.7
yacc	2.79%	5.57%	0.55%	7.13%	13.1	9.0

Table 8. Schemes to Reduce the Memory Traffic Ratio

name	0.5		0.7		1.0		1.1	
	miss	traffic	miss	traffic	miss	traffic	miss	traffic
cccp	2.60%	25.88%	3.02%	31.02%	2.86%	33.78%	3.21%	36.73%
cmp	0.06%	0.77%	0.05%	0.75%	0.05%	0.66%	0.05%	0.70%
compress	0.08%	1.05%	0.07%	1.00%	0.07%	0.99%	0.07%	1.02%
grep	0.03%	0.31%	0.02%	0.27%	0.02%	0.24%	0.02%	0.25%
lex	0.02%	0.21%	0.03%	0.32%	0.04%	0.41%	0.04%	0.41%
make	1.26%	13.75%	1.57%	18.22%	1.52%	19.77%	1.78%	23.10%
tar	0.32%	4.30%	0.27%	3.16%	0.28%	3.55%	0.32%	4.09%
tee	0.24%	2.97%	0.24%	2.99%	0.21%	3.00%	0.23%	2.95%
wc	0.02%	0.37%	0.02%	0.36%	0.02%	0.34%	0.02%	0.36%
yacc	0.65%	5.81%	0.64%	6.75%	0.55%	7.13%	0.42%	4.68%

Table 9. Effect of Code Scaling

tion as soon as the accessed location comes back from main memory.

The *partial* column in Table 8 presents the effect of loading only part of the missing block. The *avg.fetch* column shows the average transfer size (in 4-byte entities) for a cache miss. The *avg.exec* column indicates the average number of consecutive instructions (4-bytes each) used starting from a cache miss point to a taken branch or another cache miss. A comparison with the *64B* column shows that, for programs causing large memory traffic ratio, this approach can significantly reduce the memory traffic ratio at the cost of only slightly increasing the miss ratio. Note that for programs with extremely small miss ratio and memory traffic ratio, this scheme can actually increase both ratios. However, since the traffic ratios are so low for these programs that a slight increase does not have visible effect on the system performance.

4.2.3. Code Scaling Experiment

The code generated by the IMPACT-I C compiler very closely match the physical code of a fixed instruction format (32bits/instruction) RISC type processor. Different architectures have different code densities. To show that our result is more general, we will repeat the 2K, 64B block, partial loading experiment after code scaling. Code scaling simulates the effect of varying the degrees of instruction encoding. We scale the code to 0.5, 0.7 and 1.1 of its original size. The scaling affects the size of all basic blocks uniformly. The instruction size is still assumed to be 4 bytes, and therefore, the effect of code scaling is shown as changes in the number of instructions in basic blocks. For each basic block, the number of instructions is rounded to the nearest integer value.

The result, in Table 9, supports our claim that our compiler instruction layout optimization is generally applicable to many instruction sets and compilers with differing code improving ability. A richer instruction set may reduce the number of instructions to realize the intermediate form. Various code improving techniques can also change the code size. But the experiment result seems to indicate that the cache performance is rather stable.

4.2.4. Comparison with Previous Results

The effectiveness of the instruction placement optimization can be evaluated by comparing the numbers in Table 6 and Table 7 against the numbers in Table 1. To ensure that the comparison favors the conventional approach, we use the worst-case numbers in Table 6 and Table 7, the numbers for the *cccp* and the *make* programs. Our direct-mapped cache numbers are consistently better than the traditional fully associative cache numbers. In fact, the miss ratios are consistently less than half of the ones expected by Smith. If we take the average of the miss ratios across the 10 programs, the average miss ratio would be about 1/5 of Smith's design target miss ratios. The results are clearly in favor of the instruction placement optimization.

5. Conclusion

We have designed and implemented an instruction placement algorithm to improve the performance of the instruction memory hierarchy. Sequential and spatial localities are maximized by placing the instructions executed near each other in time into consecutive memory locations. Cache mapping conflicts are minimized by placing the functions with overlapping lifetime into memory locations which do not contend with each other in cache.

Using trace driven simulation, we have demonstrated that the instruction layout algorithm can efficiently exploit small, direct-mapped instruction caches with large blocks. The performance of an optimized direct-mapped instruction cache is better than Smith's target design miss ratio [28] using fully associative cache organization without code restructuring, for our ten benchmark programs. High instruction cache performance is achieved due to low miss ratio, low memory traffic ratio, and fast hardware. The effect of varying the cache design parameters (cache size, block size, block sectoring, and partial loading) has been presented. We have also measured the effect of varying the degree of instruction encoding on the instruction cache performance.

We are continuing this research in several directions. First, we are expanding the benchmark set to include more than 30 UNIX and CAD programs. This includes the programs, the library functions, and the representative input files. Second, we are conducting experiments on the instruction paging performance. The design parameters under investigation include working set size, page size, and page sectoring. Third, we are developing new performance measurement methods for the instruction memory hierarchy. With few mapping conflicts, performance measurements based on weighted call graphs could closely approximate the trace driven simulation. If the approximation proves to be accurate, we would be able to search the instruction memory hierarchy design space with billions of dynamic accesses.

References

1. J. Emer and D. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proceedings of the 11th Annual Symposium on Computer Architecture*, June 1984.
2. R. M. Russell, "The Cray-1 Computer System," *Comm. ACM*, vol. 21, no. 1, pp. 63-72, January 1978.
3. J. L. Hennessy , N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI Processor Architecture," *Proceedings of the CMU Conference on VLSI Systems and Computations*, October 1981.
4. P. Chow and M. Horowitz, "Architecture Tradeoffs in the Design of MIPS-X," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 2-5, 1987.
5. D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE Computer*, pp. 8 - 21, September, 1982.
6. W. W. Hwu and P. P. Chang, "Inline Function Expansion for Compiling C Programs," *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 21-23, 1989.
7. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, 1986.
8. J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. c-30, no.7, pp. 478-490, IEEE, July 1981.
9. A. J. Smith, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, ACM, September 1982..
10. M. D. Hill and A. J. Smith, "Experimental Evaluation of On-Chip Cache Memories," *Proceedings of the 11th Annu. Symposium on Computer Architecture*, Boston, Massachusetts, June 17-19, 1985.
11. R. J. Eickenmeyer and J. H. Patel, "Performance Evaluation of On-chip Register and Cache Organizations," *Proceedings of the 15th International Symposium on Computer Architecture*, Honolulu, Hawaii, May 30 - June 2, 1988.
12. C. L. Mitchell and M. J. Flynn, "A Workbench for Computer Architects," *IEEE Design and Test of Computers*, IEEE, Feburary 1988.
13. D. B. Alpert and M. J. Flynn , "Performance Trade-offs for Microprocessor Cache Memories ," *IEEE MICRO* , pp. 44 - 54, IEEE , August 1988.
14. S. Przybylski, M. Horowitz, and J. L. Hennessy, "Performance Tradeoffs in Cache Design," *The 15th International Symposium on Computer Architecture Conference Proceedings*, pp. 290-298, Honolulu, Hawaii, May 30 - June 2, 1988.
15. J. L. Hennessy and T. Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Trans. on Programming Languages and Systems*, vol. 5, pp. 422-448, ACM, July 1983.
16. G. Radin, "The 801 Minicomputer," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39 - 47, March 1982.
17. R. P. Colwell, R. P. Nix, J.J. O'Donnell, D. B. Papworth, P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 105-111, ACM, Palo Alto, California, October 5-8, 1987.
18. G.J. Chaitin, "Register Allocation & Spilling Via Graph Coloring," *ACM SIGPLAN Notice*, vol. 17-6, pp. 201 - 207, June 1982.
19. F. Chow and J. Hennessy, "Register Allocation by Priority-bases Coloring," *Proceedings of the ACM SIGPLAN Symposium on Compiler Constructions*, pp. 222-232, June 17-22, 1984.
20. J. R. Goodman and W.-C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," *Proceedings of the 1988 International Conference on Supercomputing*, pp. 442-452, ACM, St. Malo, France, July 4-8, 1988.
21. Ferrari, D., "Improving Locality by Critical Working Sets," *CACM*, vol. 17, no. 11, November 1984.
22. Duncan H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers*, vol. C-24, no. 12, December 1975.
23. Dennis Gannon, "Strategies for Cache and Local Memory Management by Global Program Transformation," *Journal of Parallel and Distributed Computing*, vol. 5, 1988.

24. Mauricio Breternitz Junior and John Paul Shen, "Organization of Array Data for Concurrent Memory Access," *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*, November 30-December 2, 1988.

25. Stephen J. Hartley, "Compile-Time Program Restructuring in Multiprogrammed Virtual Memory Systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 11, November 1988.

26. Scott McFarling, "Program Optimization for Instruction Caches," *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 3-6, 1989.

27. A. J. Smith, "Cache Evaluation and the Impact of Workload Choice," *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, Massachusetts, June 17-19, 1985.

28. Alan Jay Smith, "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, vol. C-36, no. 9, pp. 1063-1074, September 1987.

29. P. P. Chang and W. W. Hwu, "Trace Selection for Compiling Large C Application Programs to Microcode," *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitectures*, San Diego, California, November 29 - December 2.

Appendix: The IMPACT-I Instruction Placement Algorithm

```
MIN_PROB = 0.7;

Algorithm TraceSelection {
  /** select the best immediate successor
   ** of the basic block, bb **/
  best_successor(bb) {
    ln = the outgoing arc with the highest execution count.
    if (weight(ln)==0) return 0;
    if (weight(ln)/weight(bb) < MIN_PROB) return 0;
    if (weight(ln)/weight(destination(ln)) < MIN_PROB)
      return 0;
    if (destination(ln) has been selected) return 0;
    return ln;
  }
  /** select the best immediate predecessor
   ** of the basic block, bb **/
  best_predecessor(bb) {
    ln = the incoming arc with the highest execution count.
    if (weight(ln)==0) return 0;
    if (weight(ln)/weight(bb) < MIN_PROB) return 0;
    if (weight(ln)/weight(source(ln)) < MIN_PROB) return 0;
    if (source(ln) has been selected) return 0;
    return ln;
  }
  trace_select(F) {
    int trace_id = 0;
    if (weight(F)==0) {
      /** for non-executed functions, each basic
       ** block forms a trace.
       **/
      for (all BBi in F) {
        trace_id = trace_id + 1;
        BBi.trace_id = trace_id;
      }
      return;       /** exit function **/
    }
    /** for non-zero weight functions. **/
    sort all BBi in F according to weight(BBi);
    mark all BBi in F not-selected;
    while (there are not-selected BB) {
      trace_id = trace_id + 1;
      seed = the not-selected BB with the highest
        execution count;
      seed.trace_id = trace_id;
      /** grow the trace forward **/
      current = seed;
      for (;;) {
        ln = best_successor(current);
        if ((ln==0) or (destination(ln)==ENTRY))
          break;   /** exit for loop **/
        s = destination(ln);
        s.trace_id = trace_id;
        current = s;
      }
      /** grow the trace backward **/
      current = seed;
      for (;;) {
        if (current==ENTRY)
          break;   /** exit for loop **/
        ln = best_predecessor(current);
        if (ln==0)
          break;   /** exit for loop **/
        s = source(ln);
        s.trace_id = trace_id;
        current = s;
      }
    }
  }
}

Algorithm FunctionBodyLayout {
  mark all traces un-visited;
  function space = 0;
  current = ENTRY trace;
L1 :
  while (current<>0) {
    mark current visited;
    place the trace into the function space;
    /** try to find a connection to a trace header.
     ** we consider only non-zero weight traces.
     **/
    best = best trace connected to the current trace's
      tail. (terminal to terminal connection only)
    if (weight(best)<>0) {
      current = best;
      continue;   /* goto L1 */
    }
    /** if there is no sequential locality at all,
     ** we will start from the most important not-visited
     ** trace.
     **/
    best = the most important trace among
      not-selected traces;
    if (best==0) /** all traces have been processed. **/
      break; /* goto L2 */
    current = best;
    continue;          /* goto L1 */
L2 :
}

Algorithm GlobalLayout {
  * assume a call graph is available.
  find all call sites (Fi, Fj) == Fi calls Fj;
  weight(Fi, Fj) = sum of all calls from Fi to Fj;
    except when Fi==Fj, weight(X,X) = 0.
  for each function Fi,
    determine the size of its active region.
    determine the size of its non-active region.
  /** apply depth-first-search, mark every node **/
  Fi.visit = false for all Fi;
  from functions Fi on top of the call graph
                  hierarchy (e.g. "main")
    if (Fi.visit==false)
      Visit(Fi);
  /** layout the function according to the depth-first order. **/
  according to DFS order, layout the effective region of all
    functions.
  according to the same DFS order, layout the
    non-active region of the functions.
}

Visit(F) {
  static int id=1;
  F.visit = true;
  F.id = id++;
  sort all subcalls from F by weight(F, Fj);
  /** from the most important to the
   ** least important call site. **/
  for all callees Fj in the sorted order
    if (Fj.visit==false)
      Visit(Fj);
}
```