Control Flow Optimization for Supercomputer Scalar Processing

Pohua P. Chang and Wen-mei W. Hwu

Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Ave.
Urbana, IL 61801
pohua@bach.csg.uiuc.edu

Abstract

Control intensive scalar programs pose a very different challenge to highly pipelined supercomputers than vectorizable numeric applications. Function call/return and branch instructions disrupt the flow of instructions through the pipeline, degrading the utilization of the pipelined datapaths. This paper describes control flow optimization for scalar processing using an optimizing compiler. To obtain program control flow information, a system independent profiler has been integrated into the IMPACT-I C compiler. The control flow information obtained is converted into a weighted control graph. Based on the weighted control graph, function inline expansion, multi-way branch layout, and software branch prediction can be implemented. Using better compiler technology results in a very low cost hardware control unit (architecture) for high performance scalar processors.

1. Introduction

Pipelining Kogge Pipelined Computer increases the throughput of the instruction fetch, instruction decode, and instruction execution portions of a high performance scalar processor. Function call/return and branch instructions disrupt the flow of instructions through the pipeline, degrading the utilization of the pipelined datapaths. Procedure inline expansion is a simple compile-time code improving technique to reduce the function call/return costs, and has been implemented in many optimizing compilers. Auslander, Compiler, IBM801 Stallman GNU CC Chow Hennessy Register Allocation Huson expander Parafrase Allen Vectorization Parallelism Inline Some recent processors also provide hardware support for minimizing the extra memory accesses due to function calls. For example, the Berkeley RISC processors provide overlapping register windows to reduce the number of memory accesses required to save/restore registers and to pass parameters. Patterson Sequin VLSI RISC, September, 1982 Another example is the CRISP processor that uses stack buffers to capture the memory accesses to local variables so that register allocation crossing function calls can be simulated in hardware. Ditzel Mclellan Hardware Architecture CRISP Branch instructions can also disrupt the flow of instructions through the processor pipeline. Approximately one out of every three to five instructions is a branch instruction. McFarling Branch Emer Processor Performance, VAX The ability to reduce branch cost is therefore essential for executing control intensive scalar programs in highly pipelined scalar processors. Because of its impact on processor performance, branch handling has been studied by many researchers who have proposed many innovative techniques to sustain high instruction issue rate. Some branch handling methods will be described in the next subsection.

1.1. Previous Work

Instruction issue logic, implemented using dynamic code scheduling, have been shown to achieve substantial speedup in program execution. Weiss, Smith, Instruction Issue Patt, Hwu, Shebanow, HPS, Rationale Hwu, hpsm, exploit concurrency Acosta, Torng, Dispatch Stack These techniques provide special hardware to resolve data dependencies and to exploit fine-grain program parallelism at run-time. However, it is not clear from these studies that the reported speedup will occur when executing large and control intensive scalar programs. Because branch instructions are very frequent in scalar programs, very few instructions are exposed to the instruction issue logic before fetching a branch instruction that disrupts the instruction fetch pipeline.

Many branch handling methods have been studied. Some schemes employ hardware or software techniques to predict the direction of a branch and to fetch the target instructions of a branch, causing no or minimum slacks in the instruction issue stream. McFarling, Branch Lee, Smith, Branch Prediction Smith, Branch Prediction, June 1981 DeRosa, Branch Handling Ditzel, Branch folding, CRISP Shebanow, Patt, Autocorrelation Branch Prediction When the branch prediction is incorrect, instructions of the wrong control flow direction are squashed and the execution resumes in the correct control flow path as indicated by the result of the branch instruction. These schemes all assume that the accuracy of the branch prediction is high enough to hide the penalty of squashing and refilling the instruction issue pipeline.

Branch predictions that are made during compile time are called static predictions. The simplest scheme predicts all branches as taken, and has been reported to achieve about 65% accuracy. McFarling, Branch Emer, Clark, VAX Lee, Smith, Branch Prediction Another scheme predicts all backward conditional branches as taken and all forward branches as not-taken, and has been reported to achieve 76.5% accuracy, however in some cases, only 35% accuracy. Smith, Branch Prediction, June 1981 The reported prediction accuracy is highly dependent on the compiler technology. If loop unrolling and trace selection are used, the number of taken branches may decrease.

Dynamic branch prediction requires additional hardware which can make predictions based on the branch history. The history information must also be maintained in a high speed hardware buffer. Several dynamic prediction schemes have been studied and reported to achieve impressive performance, better than 90% accuracy. Lee, Smith, Branch Prediction Smith, Branch Prediction, June 1981 Hwu, Conte, Chang, Branches Shebanow, Patt, Autocorrelation Branch Prediction The performance of these hardware schemes may be degraded due to context swaps.

A mixed static and dynamic branch prediction method requires the use of a profiler. The dynamic behavior of a program, gathered while profiling the program, can be used by a compiler to make static branch predictions. Usually, the instruction set is modified to include a prediction bit in the branch instruction format. Through this bit, the compiler can convey the prediction decision to the hardware. For example, this approach is used in the MIPS architecture. McFarling, Branch

However, it should be noticed that branch prediction does not solve the branch problem. Even with 100% prediction accuracy, taken conditional branches still cause slacks in the instruction issue pipeline, because the branch target address can not be computed in time. The first problem is that the instruction issue logic prepares to branch only when the branch instruction has reached the end of the decoding stage, which is usually too late. Lee and Smith have studied some hardware mechanisms that deliver the branch target addresses as early as possible, by keeping the branch target address in a fast buffer which is indexed by the address of the branch instructions. Lee, Smith, Branch The second problem is that, accessing the instruction cache with the target address adds additional delay. To further reduce this delay, the instructions in the target path can also be kept in a fast buffer to avoid the instruction cache access latency. Caching the first few instructions from the target paths of frequently executed predicted-taken branches are the most beneficial.

In an attempt to reduce hardware complexity, RISC processors, including IBM 801, Radin, March 1982 Berkeley RISC-I, Patterson, RISC, VLSI Stanford MIPS, Hennessy, MIPS and HP Spectrum, Birnbaum, Beyond RISC all employ the delayed branch approach. In this approach, the compiler fills the delay slots following the branch instruction with instructions from before the branch. Regardless of the branch direction, the instructions in the delay slots are always executed. McFarling and Hennessy reported that a single delay slot can be successfully filled by the compiler for approximately 70% of the branches, and a second delay slot can be filled only 25% of the time. McFarling, Hennessy, Branch Therefore, it is hard to support moderately pipelined instruction fetch units using the delayed branch technique.

McFarling and Hennessy described a *squashing* branch scheme that allows more useful instructions to be placed after branch instructions. McFarling, Hennessy, Branch *Delayed branching with squashing* has been adopted by many recent RISC processors to improve their delayed branching performance. Hill SPUR Horowitz Chow MIPS-X Architecture MIPS R2000 Architecture Melear 88000 RISC Hwu, Conte, and Chang have studied a forward semantic scheme, which requires fewer delay slots than McFarling's squashing branch. Hwu, Conte, Chang, Branch Prediction Requiring fewer delay slots implicitly reduces code expansion.

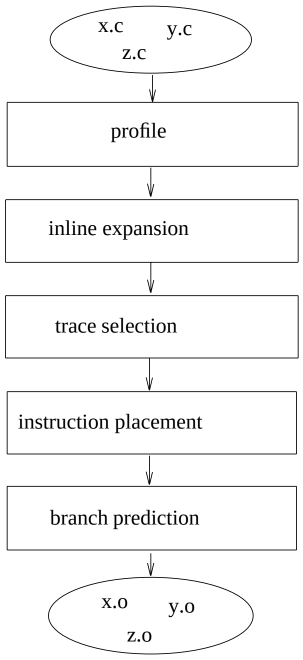
1.2. Our Approach

Figure 1 illustrates the order in which various compile-time techniques are applied in our compiler. The information gathered during profiling a program tells the compiler not only the execution frequencies of all instructions, but also the frequencies of all control transfers. In other words, for each conditional branch instruction, the compiler knows the number of times each of the possible branch directions is taken.

From the instruction execution frequencies, the invocation frequencies of each procedure call site can be deduced. With this knowledge, the compiler can selectively inline expand function bodies to obtain larger leaf procedures, and to reduce the calling overhead. Hwu, Chang, Inline

Knowing the frequencies of all control transfers, the compiler can apply program restructuring techniques, such as trace selection, Chang, Hwu, Trace Selection Ellis, Bulldog Fisher, Trace Scheduling, July 1981 to maximize instruction sequential and spatial localities Hwu, Chang, Instruction Placement and to reduce the number of taken branches.

Figure 1: Compiler Filters



After function inline expansion, trace selection, and placement, the compiler makes predictions for all two-way conditional branches based on the profile information. In addition, the compiler can intelligently implement multi-way branches, such as switch statements in C, as either a two-way branch sequence or a hashing jump.

This approach relies on compiler techniques to optimize control flow in pipelined processors. Minimal hardware is needed, only one additional bit for each conditional branch instruction. Through this additional bit, the compiler can tell the hardware control unit whether or not the branch will likely be taken. In a later section, we describe how a profiler can be integrated into a compiler, and how the profiled information can be captured in a well-defined data structure to which code improving techniques can be applied.

In this paper, we report the effectiveness of profiled branch prediction after inline expansion, trace selection, and code restructuring to reduce likely-taken branches have been performed.

1.3. Organization of Paper

Section 2 describes the program profiling utility in the IMPACT-I C compiler. Section 3 shows the result of function inline expansion. Section 4 presents a method to reduce the cost of multi-way branches. Section 5 shows the result of two-way conditional branch prediction. In section 6, we give some concluding remarks.

2. Program Profiling

A system independent execution profiler Chang, Hwu, Trace Selection has been implemented and integrated into the IMPACT-I C compiler. To make the profile information useful to the compiler, the profile information must be presented in a structure which can be easily understood by the compiler. In our C compiler, a program is represented by a weighted 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 one 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 to C compiler interface allows the profile information to be automatically used by the IMPACT-I C compiler.

To profile a C program, the IMPACT-I profiler converts the program into a functionally equivalent C program with all the probes inserted. This new C program can then be compiled by the C compilers of different systems and executed on these systems to collect profile information concurrently. 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 programs for which representative inputs can be easily collected.

We have chosen 14 commonly used scalar programs from several areas of scalar processing: program compilation, text processing, data compression, and computer-aided designing. These benchmarks all exhibit complex control structures and are good examples of high-level language programming. We expect that the scalar portions of large supercomputing applications have similar control structures. Table 1 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 experiment. The *size* column shows the number of bytes required to store the profile information. The average storage requirement is about six (6) bytes per one line of C code. The *description* column describes the benchmark programs. The inputs are realistic and representative of typical uses of the benchmarks. For example, the grammars for a C compiler and for a LISP interpreter are two of ten realistic inputs for *bison* and *yacc*. Twenty files of several production quality C programs, ranging from 100 to 3000 lines, are inputs to the *cccp* program. The *input description* column briefly describes the nature of theinputs for the benchmarks.

name	line	run	size(B)	description
bison	6913	10	44314	GNU version of yacc
cccp	4660	20	20411	GNU C preprocessor
cmp	371	16	1098	Compare text files
compress	1941	20	5086	File compression
eqn	4167	20	15860	Typeset mathematics
espresso	11545	20	49805	Logic minimization
grep	1302	20	2955	String search
lex	3251	4	26722	Lexical analyzer generator
make	7043	20	24258	Maintain files
tar	3186	14	11634	Tape archives
tbl	4497	20	30484	Format tables
tee	1063	18	1846	Replicate input
wc	345	20	1056	Word count
yacc	3333	10	29677	Parsing program generator

Table 1. Benchmark characteristics

name	input description
bison	grammar for a C compiler, etc.
cccp	C programs (100-3000 lines)
cmp	similar/text files
compress	same as cccp
eqn	papers with .EQ options
espresso	original espresso benchmarks*
grep	exercised various options
lex	lexers for C, Lisp, awk, and pic
make	makefiles for cccp, compress, etc.
tar	save/extract files
tbl	papers with .TS options
tee	text files (100-3000 lines)
wc	same as cccp
yacc	grammar for a C compiler, etc.

Table 1. (continued)

3. Eliminating Function Calls

Structured programming techniques encourage the use of functions. As a result, realistic scalar programs often execute a large number of function calls. Unfortunately, function calls cause performance problems by hindering compiler optimizations across function boundaries. Examples of compiler optimizations hindered by function calls include register allocation, code scheduling, common subexpression elimination, and constant propagation. The decreased effectiveness of these optimization techniques increases memory accesses, decreases pipeline efficiency, and increases redundant computation. These consequences are particularly magnified in supercomputers because of deep pipelining and large memory access penalties. We have implemented a function inline expansion algorithm, Chang, Hwu, Inline and obtained some interesting measurements.

Table 2 shows the static function call characteristics of the benchmarks. Each static function call corresponds to one function call in the source program. The *total* column gives the number of different function calls in the static program. Note that different static function calls could be calling the same function. We categorize the static function calls into four types. The *external* column gives the percentage of static function calls to functions whose body are unavailable to inline expansion. The *pointer* column gives the percentage of static function calls through pointers. Function calls through pointers defeat inline expansion. The *unsafe* column gives the number of static function calls which either introduce function bodies into recursive paths or have an estimated execution count less than 10. The *safe* column gives the percentage of the static functions which can be safely inline expanded. Only the safe function calls are considered for inline expansion. Only a small percentage of the static functions can be safely inline expanded.

benchmark	total	external	pointer	unsafe	safe
bison	1026	40.4%	0.0%	49.9%	9.6%
cccp	395	16.2%	0.0%	72.2%	11.4%
cmp	40	50.0%	0.0%	2.5%	47.5%
compress	192	40.6%	0.0%	58.9%	0.5%
eqn	470	5.1%	0.0%	78.5%	16.4%
espresso	1487	6.1%	0.7%	63.3%	29.9%
grep	90	20.0%	0.0%	73.3%	6.7%
lex	583	12.5%	0.0%	71.9%	15.6%
make	686	15.2%	0.0%	64.1%	20.7%
tar	446	31.4%	0.0%	64.3%	4.3%
tbl	797	4.4%	0.0%	74.8%	20.8%
tee	82	40.2%	0.0%	59.8%	0.0%
wc	27	48.1%	0.0%	51.9%	0.0%
yacc	485	22.7%	0.0%	62.3%	15.1%

Table 2. Static function call characteristics.

Table 3 presents the dynamic behavior of function calls. A dynamic function call is an invocation of a function from a particular call site. A static function call can correspond to many dynamic function calls. Only those static functions corresponding to a large number of dynamic function calls should be considered for inline expansion. Note that the small percentage (about 15%) of safe static functions accounts for a large percentage (about 70%) of all dynamic functions. The only exceptions are *wc*, which has almost no function calls, and *tee*, which has almost no function calls to user functions, and therefore are irrelevant to this discussion. This is

* See R. Rudell, "Espresso-MV: Algorithms for Multiple-Valued Logic Minimization, Proc. Cust. Int. Circ. Conf., May 1985.

encouraging since it implies that we only need to inline expand a small percentage of the function calls to eliminate a large percentage of the dynamic calls.

benchmark	total	external	pointer	unsafe	safe
bison	31.1K	36.6%	0.0%	1.3%	62.0%
cccp	3.4K	4.6%	4.2%	5.6%	85.5%
cmp	1.0K	50.2%	0.0%	0.5%	49.3%
compress	4.3K	0.4%	0.0%	0.6%	98.9%
eqn	46.6K	7.8%	0.0%	0.8%	91.4%
espresso	295.8K	0.1%	9.4%	0.2%	90.3%
grep	17.5K	1.2%	0.0%	0.1%	98.8%
lex	55.4K	6.0%	0.0%	0.7%	93.4%
make	51.7K	9.2%	0.0%	0.3%	90.5%
tar	1.3K	35.2%	0.0%	17.6%	47.2%
tbl	32.0K	14.6%	0.0%	3.4%	82.0%
tee	1.6K	99.1%	0.0%	0.9%	0.0%
wc	21	53.1%	0.0%	46.9%	0.0%
yacc	56.2K	7.7%	0.0%	0.4%	91.9%

Table 3. Dynamic function call behavior.

Table 4 offers the most important results of inline expansion. 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 *IL's per call* column gives the average number of dynamic intermediate 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.

benchmark	code inc	call dec	IL's per call	CT's per call
bison	17%	50%	630	125
cccp	22%	51%	424	73
cmp	3%	49%	254	48
compress	4%	99%	21865	2735
eqn	20%	82%	182	41
espresso	23%	70%	5986	871
grep	20%	95%	2522	844
lex	17%	85%	4624	1386
make	20%	54%	328	62
tar	13%	37%	884	110
tbl	30%	66%	55	13
tee	0%	0%	14	4
wc	0%	0%	17478	4118
yacc	21%	80%	1151	266

Table 4. Inline expansion results.

Note that inline expansion mechanism eliminates large percentages of dynamic function calls for function call intensive programs. For programs with less frequent function calls to begin with, the inline expansion mechanism does not eliminate large percentages of dynamic function calls. This is a desirable behavior because the overall goal is to ensure infrequent function calls rather than to achieve high elimination percentages.

All in all, the inline expansion is very effective in that function calls only account for less than 1% of the control transfers after inline expansion (see the *CT's per call* column) for computation intensive programs (espresso, lex, yacc). In terms of frequency, several hundreds of dynamic instructions are executed between dynamic function calls for computation intensive programs. Therefore, function calls become unimportant in the hardware design considerations. Also, large scopes for compiler optimizations can be expected for the critical parts of the programs. The price, on the average, is a 15% increase in static code size. From a first look, this may cause instruction cache/buffer performance degradation. However, in a previous paper, we demonstrated that inline expansion trades code expansion for reduced mapping conflicts and increased sequentiality. The net results prove that inline expansion actually improves the instruction cache/buffer performance rather than degrades it.

4. Reducing the Cost of Multiway Branches

The distribution of various types of branch instructions is listed in Table 5. The *%condition* column of Table 5 indicates the percentage of conditional branch instructions among all the dynamic control transfer instructions. The *%uncondition* column of Table 5 indicates the percentage of unconditional branch (including call/return) instructions among all the dynamic control transfer instructions. Note that the effects of inline expansion to reduce function calls is included in this number. The *%multi-way* column of Table 5 indicates the percentage of multi-way branch instructions among all dynamic control transfer instructions. Although the percentage of multi-way branch instructions is small, they are nevertheless important due to their long potential execution time.

name	%condition	%uncondition	%multi-way
bison	92.8%	6.8%	0.3%
cccp	69.0%	11.0%	19.9%
cmp	80.5%	19.4%	0.0%
compress	90.5%	9.5%	0.0%
eqn	91.5%	7.4%	1.0%
espresso	85.7%	13.5%	0.9%
grep	82.2%	13.3%	4.4%
lex	98.4%	1.5%	0.1%
make	93.7%	6.0%	0.3%
tar	97.2%	2.8%	0.0%
tbl	81.4%	17.8%	0.8%
tee	79.6%	20.4%	0.0%
wc	91.4%	8.6%	0.0%
yacc	97.1%	2.7%	0.2%

Table 5. Percentage of various branch types

Each multi-way branch (switch statement) can be implemented by either a hashing jump or a sequence of conditional branches. The IMPACT-I C compiler implements each multi-way decision as follows. First, the compiler sorts all the target cases by their probability of execution. Second, the compiler lays out the conditional branches so that the ones with higher branching probability appear before those with lower branching probabilities. An exception to this rule is the *default* case which has to be placed at the very end as an unconditional jump instruction. Third, the compiler calculates the expected number of comparisons to implement the multi-way decision with the sequence of conditional branches formed in the second step. If the expected number of comparisons is beyond a threshold (10 in this measurement), a hashing jump will be used instead. The execution of these hashing jumps involves hashing the input condition into a hash table of explicit and default cases, fetching the corresponding target address, and redirecting the instruction fetch with that target address.

Table 6 shows the results of the multi-way branch implementation. The *%default* column indicates the percentage of the time the *default* case is reached for all switch statements. For some benchmarks, the *%default* percentage is high due to the low coverage of the explicit cases. Because we must place the *default* case at the end of the branch sequence as an unconditional branch instruction, high *%default* percentage lessens the effectiveness of compiler case layout optimization. The effect is especially pronounced in *eqn*.

The *%hashing* column indicates the percentage of all multi-way branches being implemented by hashing jumps. For architectures with long scalar memory access delays, the threshold for adopting the hashing jumps could be increased to much more than the one we used (10 expected comparisons). Therefore, one can expect to see a smaller percentage of hashing jumps for architectures with long scalar memory delays.

The *%sequence* column of Table 6 indicates the percentage of all switch statements being implemented by branch sequences. The *total* column indicates the average number of cases per multi-way branch implemented by branch sequences, excluding the default case. The *expected* column indicates the expected number of comparisons required to resolve a multi-way branch implemented as a branch sequence. Note that for most benchmarks, the sequence number is close to 100%. For *compress, grep, and lex,* the high percentage of branch sequence implementation is due to the highly biased distribution of selecting cases. For these benchmarks, the average total number of comparisons is high (≥ 10) but the expected number of comparisons is much lower (≤ 5). For the other benchmarks, almost all multi-way branches are implemented as branch sequences, due to their small numbers of total cases.

name	**%default**	**%hashing**	**%sequence**	**total**	**expected**
bison	74.7%	9.3%	90.7%	6.96	6.22
cccp	92.8%	51.8%	48.2%	3.36	3.15
cmp	0.0%	0.0%	100.0%	3.00	1.00
compress	0.0%	0.0%	100.0%	10.00	1.00
eqn	84.0%	75.3%	24.7%	6.97	6.13
espresso	66.2%	0.0%	100.0%	2.71	2.00
grep	0.0%	0.0%	100.0%	12.00	1.50
lex	35.9%	0.0%	100.0%	12.72	5.42
make	39.7%	0.0%	100.0%	8.60	4.56
tar	0.0%	0.0%	100.0%	6.38	1.26
tbl	22.4%	0.0%	100.0%	11.99	2.94
tee	0.0%	0.0%	100.0%	3.00	1.00
wc	0.0%	0.0%	100.0%	3.00	1.60
yacc	48.5%	0.0%	100.0%	6.28	4.87

Table 6. Multiway branch statistics

Assuming a hashing jump is 10 times more expensive than each conditional branch, the cost of each multi-way branch has been reduced to about 3.5 conditional branches per multi-way branch. Considering the low dynamic percentage of multi-way branches among all control transfers, we conclude that the cost of multi-way branches is no longer a major concern.

5. Reducing the Cost of Conditional Branches

This section examines the characteristics of the conditional branches corresponding to the two-way decisions in C programs. These branches are due to *if* statements, the conditional operators (&& || ?:), and the loop control structures. The IMPACT-I C compiler uses the profile information to lay out the instruction space to reduce the frequency of taken branch instructions. For each function, basic blocks which tend to execute in sequence are grouped into traces. The traces form the basic units of instruction placement to minimize the dynamic frequency of taken branches. The process of grouping basic blocks into traces is called *trace selection*. Fisher Trace Scheduling Compaction A detailed description and evaluation of the trace selection algorithm of the IMPACT-I C compiler is given in a recent paper. Chang Hwu Trace Selection

Table 7 presents the trace selection results. Note that inline expansion provides large functions to enhance the size of the traces selected. The *desirable* column gives the percentage of control transfers which go from a basic block to its successor within a trace. The *undesirable* column gives the percentage of control transfers which enter and/or exit traces at a non-terminal basic block. The small average percentage (about 3%) in the *undesirable* column and the large average percentage (about 61%) 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 traces as units of instruction placement. By using traces as the units of the instruction placement algorithm, we have at least 61% of dynamic branches not taken. This number is further increased by placing traces carefully into a linear order. Note that the percentage of not taken branches due to intra-trace sequentiality alone is significantly higher than the traditionally reported 33% number. Emer Clark Characterization VAX 780 Smith Lee Target Buffer

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 36%) for this category suggests that loop unrolling and a careful selection of linear ordering of traces could significantly reduce the dynamic frequencies of taken branches. We start with the entrance trace, and expand the placement by placing the most important descendant 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.

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), a unit of instruction placement contains about 54 bytes. Considering the spatial locality among traces, a reasonable prediction for a good instruction buffer bank size would be 64 bytes.

name	neutral	undesirable	desirable	trace length
bison	39.4%	1.4%	59.2%	2.5
cccp	49.9%	3.7%	46.4%	1.9
cmp	12.7%	4.2%	83.0%	6.9
compress	35.1%	2.9%	62.0%	2.8
eqn	15.7%	2.0%	82.3%	5.9
espresso	49.3%	5.4%	45.3%	1.9
grep	21.3%	1.4%	77.4%	4.6
lex	35.2%	1.8%	63.1%	2.8
make	54.2%	2.1%	43.7%	1.8
tar	88.0%	0.5%	11.5%	1.1
tbl	19.6%	2.4%	78.0%	4.3
tee	24.8%	0.2%	75.0%	4.0
wc	15.1%	9.0%	75.9%	5.5
yacc	49.2%	4.6%	46.2%	2.0

Table 7. Trace selection results

Table 8 shows a detailed breakdown of the statically predicted and actual behavior of branches. Column *TT* of Table 8 indicates the number of branches which are predicted to be taken and are actually taken, as a percentage of all conditional branches. Column *TN* of Table 8 indicates the number of branches which are predicted to be taken but are actually not taken, as a percentage of all conditional branches. Column *NT* of Table 8 indicates the number of branches which are predicted not taken but are actually taken, as a percentage of all conditional branches. Column *NN* of Table 8 indicates the number of branches which are predicted not taken and are actually not taken.

name	TT	TN	NT	NN	hit_ratio
bison	33.4%	3.1%	5.4%	58.1%	91.5%
cccp	42.5%	6.0%	5.2%	46.3%	88.8%
cmp	0.0%	0.0%	3.1%	96.9%	96.9%
compress	18.3%	2.8%	11.5%	67.4%	85.7%
eqn	14.2%	3.3%	3.3%	79.2%	93.4%
espresso	26.5%	6.3%	9.2%	58.0%	84.5%
grep	8.3%	0.3%	1.7%	89.7%	98.0%
lex	46.6%	1.1%	1.7%	50.6%	97.2%
make	49.8%	3.3%	2.5%	44.4%	94.2%
tar	90.2%	0.7%	0.6%	8.6%	98.8%
tbl	24.5%	1.7%	3.7%	70.1%	94.6%
tee	12.3%	0.1%	12.7%	75.0%	87.3%
wc	10.6%	2.9%	11.2%	75.3%	85.9%
yacc	38.6%	2.0%	8.1%	51.3%	89.9%

Table 8. Conditional branch results

Two observations are worth mentioning. First, about 65% of the dynamic branches are not taken and almost all of them can be correctly predicted at the compile time. Comparing this number with the traditional 33% number shows that our instruction placement algorithm is effective in reducing taken branches. Second, among the taken branches (which account for about 35% of the dynamic branches), most of them can also be correctly predicted at the compile time. Overall, about 92% of the dynamic branches can be correctly predicted at the compile time. Supercomputer architectures which take advantage of this compilation support can have a significant advantage over those which do not take advantage of it.

6. Conclusion

We have developed a sequence of profile-based control flow optimizations. These optimizations reduce the negative effects of function calls and branches on the supercomputer scalar performance. These optimization techniques have been implemented in our IMPACT-I C compiler and evaluated with a set of realistic UNIX domain programs. The results presented in this paper are derived from the real execution of several billions of dynamic instructions. The size of the experiment is significantly larger than most previously reported results.

With inline expansion, we have reduced the function call frequencies to about 1% of all the control transfers. On the average, several hundred dynamic instructions are executed between dynamic function calls. This eliminates function calls as a major issue of scalar processing.

With a careful implementation of multi-way branches, we have reduced the cost of multi-way branches to about 3.5 times that of a conditional branch. With a low percentage of multi-way branches (about 10% out of all control transfers), this leads us to believe that multi-way branches have been removed from being a major issue of scalar processing.

Conditional and unconditional branch instructions remain as a critical issue of scalar processing. About 20% of the dynamic instructions are branches. By taking advantage of the profile information, the IMPACT-I instruction placement has reduced the percentage of the dynamic taken branches from 67% to 35%. This improves the efficiency of the instruction pipeline and the instruction cache/buffer. Meanwhile, we showed that 92% of the dynamic branches can be predicted correctly at the compile time. Therefore, supercomputer architectures can significantly improve their scalar performance by taking advantage of the compile-time branch prediction.

Acknowledgements

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.

The authors would like to acknowledge Sadun Anik, Nancy Warter, Thomas Conte, and the other members of the Computer System Group for their invaluable comments and suggestions.

$LIST$