In Search of Near-Optimal Optimization Phase Orderings

Prasad A. Kulkarni David B. Whalley
Gary S. Tyson
Florida State University
Computer Science Department
Tallahassee, FL 32306-4530
{kulkarni,whalley,tyson}@cs.fsu.edu

Jack W. Davidson
University of Virginia
Department of Computer Science
Charlottesville, VA 22904-4740
jwd@virginia.edu

Abstract

Phase ordering is a long standing challenge for traditional optimizing compilers. Varying the order of applying optimization phases to a program can produce different code, with potentially significant performance variation amongst them. A key insight to addressing the phase ordering problem is that many different optimization sequences produce the same code. In an earlier study, we used this observation to restate the phase ordering problem to concentrate on finding all distinct function instances that can be produced due to different phase orderings, instead of attempting to generate code for all possible optimization sequences. Using a novel search algorithm we were able to show that it is possible to exhaustively enumerate the set of all possible function instances that can be produced by different phase orderings in our compiler for most of the functions in our benchmark suite [1]. Finding the optimal function instance within this set for almost any dynamic measure of performance still appears impractical since that would involve execution/simulation of all generated function instances. To find the dynamically optimal function instance we exploit the observation that the enumeration space for a function typically contains a very small number of *distinct control flow paths*. We simulate only one function instance from each group of function instances having the identical control flow, and use that information to estimate the dynamic performance of the remaining functions in that group. We further show that the estimated dynamic frequency counts obtained by using our method correlate extremely well to simulated processor cycle counts. Thus, by using our measure of dynamic frequencies to identify a small number of the best performing function instances we can often find the optimal phase ordering for a function within a reasonable amount of time. Finally, we perform a case study to evaluate how adept our genetic algorithm is for finding optimal phase orderings within our compiler, and demonstrate how the algorithm can be improved.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - compilers, optimization; D.4.7 [*Organization and Design*]: Real-time systems and embedded systems

General Terms Performance, Measurement, Algorithms.

LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada.

Keywords Phase Ordering, Exhaustive Search, Genetic Algorithms.

1. Introduction

Optimizing compilers employ numerous optimization phases to improve the performance of the generated program. The performance criteria is frequently speed of execution, but it may also be code size or energy consumption for applications compiled for embedded processors. It is commonly acknowledged that the code generated by even the best optimizing compilers can almost always be improved. One reason for sub-optimal code being produced is due to an issue in compilers called the *phase ordering problem*.

Optimization phases in a compiler use and share resources and most require specific conditions in the code to be applicable, which can be created or destroyed by other optimizations. As a result these phases can potentially enable or disable opportunities for other optimizations. Most of these phase interactions are difficult to predict as they depend on the function being optimized, the underlying architecture, and the specific implementation of optimizations in the compiler. Moreover, a single order of applying optimizations is widely acknowledged to not produce optimal code for every application [2, 3, 4, 5, 6, 7]. Therefore, researchers have been trying to understand the properties of the optimization phase order search space so that optimal or near-optimal optimization sequences can be applied when compiling applications.

An obvious solution to the phase ordering problem is to exhaustively evaluate all orderings of optimization phases. Until recently this was considered infeasible, except for very small kernels and very few optimization phases. The reason for this can be easily appreciated when one considers the fact that most modern compilers have numerous optimization phases, and many phases are applicable multiple times in the compilation process. It is also difficult to employ reasonable heuristics to prune the optimization space that do not also prune optimal orderings, since the interactions between different optimizations are poorly understood. During one recent investigation of the phase ordering problem [5, 8] we showed that there is much redundancy in the function instances generated by different optimization phase sequences. This redundancy occurs when phases fail to transform the code or when transformations performed by different phases are independent. This key insight allowed us to approach the phase ordering problem differently, as the problem of generating all distinct function instances that can be produced by different phase orderings in our compiler. Using an innovative algorithm we were able to exhaustively enumerate all distinct function instances that can be produced by different phase orderings performed by our compiler for more than 98% of the functions in our benchmark suite [1].

The above study enumerated the optimization phase order space and determined the statically (code size) optimal function instance that can be generated by our compiler. It is, however, more challenging to determine which one of the enumerated function instances provides the best performance for dynamic performance criteria. Obtaining dynamic performance measures traditionally requires execution or simulation of the application. Executing the program for each distinct function instance takes considerably longer than the compiler generating the function instance. Moreover, simulation can be orders of magnitude more expensive than native execution, and is sometimes the only resort for evaluating the performance of applications on embedded processors. Very often embedded application developers are willing to wait for reasonably long searches, typically spanning a few hours, for small performance improvements, but most people would be unwilling to wait longer. In this paper we demonstrate the application of a technique that requires only a few executions of the program to get dynamic performance measures for all distinct function instances. Thus, the major contributions of this work are:

1. an approach to quickly obtain dynamic performance measures for all distinct function instances in an exhaustive enumeration of the optimization phase order search space;
2. a study to show that our estimates of dynamic frequency counts correlate very closely with simulator cycles for embedded systems, allowing near optimal phase orderings to be obtained;
3. a study of how adept a basic genetic algorithm is for finding the optimal phase ordering for our compiler.

The remainder of this paper is organized as follows: In Section 2 we review a summary of some other work related to this topic. In Section 3 we give an overview of our experimental framework. We provide a background to this work in Section 4 by describing our approach [1] to enumerate all distinct function instances produced by our compiler. We describe our algorithm to obtain dynamic measures for all function instances in Section 5. In Section 6 we analyze the correlation between our measure of dynamic counts and processor cycles. In Section 7 we show how well suited a genetic algorithm is to search for good optimization sequences in our compiler. In the last two sections we list some of our directions for future work and our conclusions.

2. Related Work

Several groups have worked on addressing the phase ordering problem, and attempted to better understand optimization phase interactions. Specifications of code improving transformations have been automatically analyzed to determine if one type of transformation could enable or disable another [9, 3]. This work was limited by the fact that in cases where phases do interact, the determination of the order-dependent phases was not possible to be automated and required detailed knowledge of the compiler. We have recently demonstrated that exhaustive enumeration of all distinct function instances that can be produced due to different phase orderings is possible over the entire set of optimizations in a typical compiler, and does not require more than a few hours for even the largest function in our test suite [1]. Zhao et al. [10] used a model-based framework to selectively apply optimizations when they were predicted to be profitable. This work shows that model-based approaches can be fast and accurate, and can be effectively used to explore different properties of compiler optimizations.

Researchers have also investigated the problem of finding an effective optimization phase sequence by aggressive pruning and/or evaluation of only a portion of the search space. A method, called Optimization-Space Exploration [7], uses static performance estimators to reduce the search time. In order to prune the search they limit the number of configurations of optimization-parameter value pairs to those that are likely to contribute to performance improvements. This area has also seen the application of other search techniques to *intelligently* search the optimization space. Hill climbers [11, 6] and grid-based search algorithms [12] have been employed during iterative algorithms to find optimization phase sequences better than the default one used in their compilers. Other researchers have used genetic algorithms [4, 5] with aggressive pruning of the search space [8, 13] to make searches for effective optimization phase sequences faster and more efficient.

A related issue to the phase ordering problem is how to best apply a given optimization phase. The small area of the transformation space formed by applying loop unrolling (with unroll factors from 1 to 20) and loop tiling (with tile sizes from 1 to 100) was analyzed for a set of three program kernels across three separate platforms [14]. Enumerations of search spaces formed by a larger set of distinct optimization phases have also been evaluated [11]. One important deduction was that the search space generally contains enough local minima that biased sampling techniques, such as hill climbers and genetic algorithms, should find good solutions. Rather than changing the order of optimization phases, several researchers work on a related problem of finding the best set of optimizations by turning on or off optimization flags to a conventional compiler [15, 16, 17].

Studies of using static performance estimations to avoid program executions have been done previously [18, 19, 20]. Wagner et al. [19] presented a number of static performance estimation techniques to determine the relative execution frequency of program regions, and measured their accuracy by comparing them to profiling. They found that in most cases static estimators provided sufficient accuracy for their tasks. Knijnenburg et al. [18] used static models to reduce the number of program executions needed by iterative compilation. The method of static performance estimation we use in this paper is most similar to the approach of *virtual execution* used by Cooper et al. [20] in their ACME system of compilation. In the ACME system, Cooper et al. strived to execute the application only once (for the un-optimized code) and then based on the execution counts of the basic blocks in that function instance and careful analysis of transformations applied by their compiler they tried to determine the dynamic instruction counts for all other function instances. Due to this ACME has to maintain detailed state, which introduces some amount of additional complexity in the compiler. In spite of this, in a few cases ACME is not able to accurately determine the dynamic instruction count, resulting in some error in their computation.

3. Experimental Framework

The work in this paper attempts to demonstrate that it is possible to extend the exhaustive phase order enumeration algorithm [1] to determine the dynamically optimal function instance that can be generated by a typical compiler without significant additional overhead. The research in this paper uses the Very Portable Optimizer (VPO) [21], which is a compiler back end that performs all its optimizations on a single low-level intermediate representation called RTLs (Register Transfer Lists). Since VPO uses a single representation, it can apply most analysis and optimization phases repeatedly and in an arbitrary order. VPO compiles and optimizes one function at a time. This is important for the current study since restricting the phase ordering problem to a single function, instead of the entire file, helps to make the optimization phase order space more manageable. Similar to the earlier study [1], we have used the compiler to generate code for the StrongARM SA-100 processor using Linux as its operating system. We used the SimpleScalar set of functional and cycle-accurate simulators [22] for the ARM to get dynamic performance measures.

Table 1 describes each of the 15 candidate code-improving phases that were used during the exhaustive exploration of the optimization phase order search space. In addition, *register assignment*, which is a compulsory phase that assigns pseudo registers to hardware registers, must be performed. VPO implicitly performs register assignment before the first code-improving phase in a sequence that requires it. Two other optimizations, *merge basic blocks* and *eliminate empty blocks*, were removed from the optimization list used for the exhaustive search since these optimizations only change the internal control-flow representation as seen by the compiler and do not directly affect the final generated code. These optimizations are now implicitly performed after any transformation that has the potential of enabling them. After applying the last code-improving phase in a sequence, VPO performs another compulsory phase that inserts instructions at the entry and exit of the function to manage the activation record on the run-time stack. Finally, the compiler also performs predication and instruction scheduling before the final assembly code is produced. These last two optimizations should only be performed late in the compilation process, and so are not included in the set of phases used for exhaustive optimization space enumeration. While most of the optimizations in Table 1 can be performed in an arbitrary order, a few restrictions are imposed in the compiler to simplify its implementation [1].

In this study we are only investigating the phase ordering problem and do not vary parameters for how phases should be applied. For instance, we do not attempt different configurations of loop unrolling, but always apply it with a loop unroll factor of two since we are generating code for an embedded processor where code size can be a significant issue. Note that VPO is a compiler backend. Many other optimizations not performed by VPO, such as loop tiling/interchange, inlining, and some other interprocedural optimizations, are typically performed in a compiler frontend, and so are not present in VPO. We also do not perform ILP (frequent path) optimizations since the ARM is typically a single issue processor and ILP transformations would be less beneficial. In addition, frequent path optimizations require a profile-driven compilation process that would complicate this study.

We used the same benchmarks as in our phase order space enumeration study [1]. These include a subset of the *MiBench* benchmarks, which are C applications targeting specific areas of the embedded market [23]. One benchmark was used from each of the six categories of applications (a total of 111 functions). Table 2 contains descriptions of these programs.

Optimization Phase	Gene	Description
branch chaining	b	Replaces a branch or jump target with the target of the last jump in the jump chain.
common subexpression elimination	c	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
remove unreachable code	d	Removes basic blocks that cannot be reached from the function entry block.
loop unrolling	g	To potentially reduce the number of comparisons and branches at runtime and to aid scheduling at the cost of code size increase.
dead assignment elimination	h	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	i	Removes a jump by reordering blocks when the target of the jump has only a single predecessor.
minimize loop jumps	j	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	k	Uses graph coloring to replace references to a variable within a live range with a register.
loop transformations	l	Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by loop nesting level.
code abstraction	n	Performs cross-jumping and code-hoisting to move identical instructions from basic blocks to their common predecessor or successor.
evaluation order determination	o	Reorders instructions within a single basic block in an attempt to use fewer registers.
strength reduction	q	Replaces an expensive instruction with one or more cheaper ones. For this version of the compiler, this means changing a multiply by a constant into a series of shift, adds, and subtracts.
reverse branches	r	Removes an unconditional jump by reversing a conditional branch when it branches over the jump.
instruction selection	s	Combines pairs or triples of instructions together where the instructions are linked by set/use dependencies. Also performs constant folding and checks if the resulting effect is a legal instruction before committing to the transformation.
remove useless jumps	u	Removes jumps and branches whose target is the following positional block.

Table 1. Candidate Optimization Phases Along with their Designations

Category	Program	Description
auto	bitcount	test processor bit manipulation abilities
network	dijkstra	Dijkstra's shortest path algorithm
telecomm	fft	fast fourier transform
consumer	jpeg	image compression and decompression
security	sha	secure hash algorithm
office	string-search	searches for given words in phrases

Table 2. MiBench Benchmarks Used in the Experiments

4. Background

In this section we outline our approach [1] to exhaustively enumerate the optimization phase order search space for the optimizations present in VPO. An important realization from past work on the phase ordering problem is that there is much redundancy in the function instances produced by different attempted optimization phase orders [1]. First, some phases are unsuccessful when attempted. Second, even though optimization phases interact with each other by virtue of sharing resources or by changing the instruction patterns in the program to enable or disable other optimizations, it is frequently the case that many optimizations can be applied independently of each other most of the time. As a result many different orderings of optimization phases generate the same code. The number of distinct function instances that can be produced by different phase orderings for any function is orders of magnitude smaller than the actual number of distinct attempted phase orderings. This can also be seen from the high amount of redundant sequences found by different heuristic algorithms [5, 8].

In view of these observations, instead of attempting to enumerate all distinct optimization phase sequences, the phase ordering problem can be made more practical by attempting to generate all distinct function instances that could be produced due to different optimization phase orderings. It is also important to note that the ability to correctly form all different phase orderings is inherently limited by the lack of knowledge of the correct *sequence length*

for each function. Optimization sequence lengths vary by function since optimizations may be always unsuccessful for some functions, while be successful one or more times for others.

Although re-stating the phase ordering problem makes it easier to find a tractable solution, designing and implementing the solution is by no means trivial. Our solution to this problem [1] breaks the phase ordering space into multiple levels, as shown in Figure 1. At the root (level 0) we start with the unoptimized function instance. For level 1, we generate the function instances produced by an optimization sequence length of 1, by applying each optimization phase individually to the base unoptimized function instance. After producing each function instance we check to see if it is identical to some other previously encountered function instance by calculating and comparing the 32 bit CRC [24] hash function values for each function. It is also likely that many optimizations do not find any opportunity to make changes to the current function instance. We refer to such unsuccessful phases as being *dormant* at that point. In contrast, phases that cause changes are called *active*. Thus, if the optimization phase was dormant, then we can be certain that the function instance has been left unmodified and so no check is required for such cases.

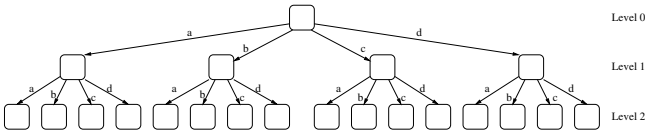

Figure 1. Naive Optimization Phase Order Space (for Four Distinct Optimizations)

For all subsequent levels, we try to generate new function instances by applying each optimization phase to all the distinct function instances present after the preceding level. This exponential space is typically reduced to a manageable Directed Acyclic Graph (DAG) due to our pruning criteria, as shown in Figure 2. We kept the enumeration times manageable by placing a constraint on their search algorithm to abort the enumeration if the number of optimization sequences to attempt at any level exceed a million. The search space for such functions is considered too large to completely enumerate in a reasonable amount of time. Even with this constraint our algorithm is able to completely enumerate the phase order space for 109 out of their 111 benchmark functions, most in a few minutes and the remainder in a few hours.

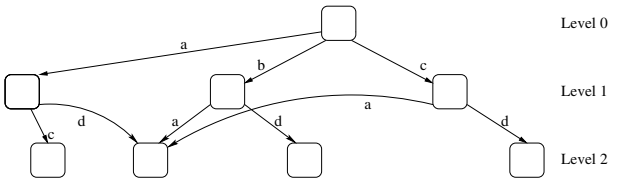

Figure 2. Optimization Phase Order Space after Applying Pruning to Eliminate Redundant Function Instances

5. Optimal Dynamic Function Instance

Exhaustive enumeration of the optimization phase order space, as explained in the previous section, can give a statically smallest (*optimal*) function instance, with respect to the possible phase orderings in our compiler. It should be kept in mind that a different compiler, with a different or greater set of optimization phases can possibly generate better code than the *optimal* instance produced by VPO. Thus, *optimal* in the context of this work refers to the best code that can be produced by any optimization phase ordering in VPO and is not meant to imply a universally optimum solution.

It is obvious that the statically smallest function instance might not produce the best possible dynamic performance. Finding the optimal code for dynamic performance measures would traditionally require execution or simulation of all distinct function instances for each function. Execution of the application can be many times more expensive than the compiler generating the function instance. Simulation is generally orders of magnitude more expensive than native execution. Our experimental environment for the ARM requires us to do simulations rather than native executions. This is a common development environment for many embedded applications since an embedded processor often is not available or does not support compilation. Simulations of all enumerated function instances in our benchmark suite may take many months or even years to accomplish. Therefore, we need some way to limit the number of simulations and still be able to accurately estimate dynamic performance for non-simulated function instances.

Our strategy to reduce the number of simulations is related to a technique used by Cooper et al. in their ACME system of adaptive compilation [20]. The concept behind our approach is based on the premise that two different function instances having identical control flow graphs will execute the same blocks the same number of times. This, together with our observation that the compiler typically generates a very small number of distinct control flow paths during the complete enumeration of all instances for any one function helped guide us to our method. Thus, we only simulate the application when the compiler produces a function instance with a control flow that has not been previously encountered. This approach of only executing a function instance when a new control flow has been encountered is more practical in VPO, which tunes applications on a per-function basis, as compared to the ACME system which tunes entire applications or each file within the application. It is difficult to compare our approach with that implemented in the ACME system since that would require us to adopt their heuristics for how basic block execution counts are to be estimated with control flow changes. Our approach is, however, easier (and provides more accurate estimates) since we always execute the application on a control flow change and so, are never required to estimate the block execution counts. It is also more practical for our experiments since we typically have very few distinct control flows per function.

Thus, the control flow graph of each new function instance is compared with all previously encountered control flows. This check compares the number of basic blocks, the position of the blocks in the control flow graph, the positions of the predecessors and successors of each block, and the relational operator of each condition branch instruction. Loop unrolling introduces a small complexity for loops with a single basic block. It is possible for loop unrolling to unroll such a loop and change the loop exit condition. Later if some optimization coalesces the unrolled blocks, then the control flow looks identical to that before unrolling, but due to different loop exit conditions, the block frequencies are actually different. Such cases are handled by marking unrolled blocks differently than non-unrolled code. We are unaware of any other control flow changes caused by our set of optimization phases that would be incorrectly detected by our algorithm.

We instrument the function instance having a *new* control flow with added instructions using EASE [25] to count the number of times each basic block in that control flow is executed. The basic block execution counts for this new control flow are recorded after simulation of the application. Other function instances having identical control flow will have identical block execution counts. We statically determine the number of cycles required to execute each basic block, which, in addition to the instruction latency,

also considers instruction-dependency and resource-conflict related stalls. Multiplying the number of static cycles for each basic block with the block execution counts gives us, what we call, the *dynamic frequency* measure for each function instance.

Table 3 shows the dynamic frequency results for all the executed functions in our benchmark suite, except for two functions in *fft* for which we were not able to completely enumerate the optimization phase order space within the constraints placed on the search algorithm, as explained in Section 4. Note that the remaining functions not shown in the table were not executed with the input data provided with these benchmarks. One important number, which makes the technique of exhaustive enumeration possible, is the relatively small number of distinct function instances for each function. For an average sequence length of 12 the number of possible attempted optimization phase orderings would be 15^{12}, where 15 is the number of distinct phases used in our study. Even more important is the fact that unlike the attempted optimization phase ordering space, the number of distinct function instances does not typically increase exponentially as the sequence length increases. Thus, even for a maximum sequence length of 26 that was encountered in the 39 executed functions we enumerated, the total number of distinct function instances is only 343,162.

It can also be seen from the table that each function typically has a very small number of distinct control flows. This fact is very important in our case, since for each function we are only required to simulate the application a small number of times. Requiring only a few simulations makes it possible to get dynamic frequency measures for all the generated function instances during the exhaustive enumeration of each function. The next column gives a count of the number of *leaf* function instances. These are function instances for which no additional optimization phase found any opportunity to be successful. The small number of leaf function instances imply that even though the enumeration DAG (see Figure 2) may grow out to be very wide, it generally starts converging towards the end. It is also worthwhile to note that it is most often the case that the optimal function instance is a leaf. Note again that we are defining optimal in this context to be the function instance that results in the best dynamic frequency measures. We found that for 87.18% of the functions in our test suite the optimal function instance is a leaf instance, with almost one in every three leaf function instances on average, yielding the same optimal dynamic performance. Apart from this, there was at least one leaf function instance having dynamic performance within 1% of the optimal for 37 out of the 39 executed functions in Table 3. It is easy to imagine why this is the case, since all optimizations are designed to improve performance, and there is no optimization in VPO which undoes changes made by other optimizations before it.

We analyzed the few cases for which none of the leaf function instances achieved optimal performance. The most frequent reason we observed that caused such behavior is illustrated in Figure 3. Figure 3(a) shows a code snippet which yields the best performance and 3(b) shows the same part of the code after applying *loop invariant code motion*. $r[0]$ and $r[1]$ are passed as arguments to both of the called functions. Thus, it can be seen from Figure 3(b) that *code motion* moves the invariant calculation, *r[4]+28*, out of the loop, replacing it with a register to register move, as it is designed to do. But later, the compiler is not able to collapse the reference by *copy propagation* due to the fact that it is passed as an argument to a function. The implementation of *code motion* in VPO is not robust enough to detect that the code will not be further improved. In most cases, this situation will not have a big impact, unless this loop does not typically execute many iterations. It is also possible that *code motion* may move an invariant calculation out of a loop that is never entered during execution. In such cases, no leaf function instance is able to achieve optimal phase ordering dynamic results.

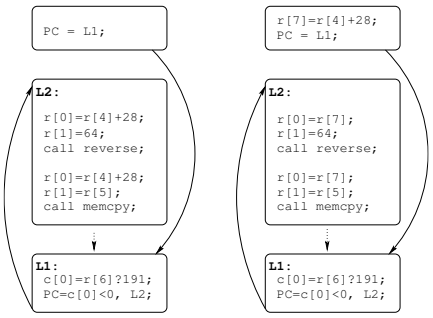

Figure 3. Case When No Leaf Function Instance Yields Optimal Performance

The two columns under the heading *% from opt.* in Table 3 show the percentage distance between the optimal function instance and the *Batch* and *Worst* leaf instances, respectively. It is important to note that in VPO the fixed (batch) sequence is always a leaf, since the batch compiler iteratively attempts optimization phases in a specific order until no additional changes are made to the function being compiled by any optimization phase. In contrast, many other compilers (including GCC) cannot do this effectively since it is not possible to easily re-order optimization phases in these compilers. The order in which phases are applied in the batch compiler has also been tuned over many years to obtain good performance. In spite of this, the batch compiler produces code which is 4.60% worse, on average, than optimal and 50% worse than optimal in the worst case. The next column shows the range between the best and the worst leaf function instance. On average this range is 47.64%.

The last three columns illustrate the percentage of leaf function instances achieving optimal or near-optimal phase order performance. Thus, we can see that more than 30% of the leaf instances yield the same optimal performance. More than 42% of the leaf function instances come within 2% of optimal. From this discussion it follows that if we can design a compiler that can generate all of the leaf function instances quickly for a function, then it is very likely that at least one of those function instances will achieve optimal or very close to optimal dynamic performance.

6. Correlation between Dynamic Frequency Measures and Processor Cycles

Our estimate of dynamic performance is partly based on static measures. Although we have tried to account for data hazards related stalls in our estimate, it still does not consider other penalties encountered during execution, such as branch misprediction and cache miss penalties. Processor cycles obtained from a simulator would provide a more accurate estimate of dynamic performance, but is also more expensive to obtain. Note that, unlike general-purpose processors, the cycles obtained from a simulator can often be very close to executed cycles in an embedded processor since these processors may have simpler hardware and no operating system. For similar reasons, dynamic frequency measures on embedded processors will also have a much closer correlation to simulated cycles, than for general-purpose processors. In this section we show that within our test environment and experimental framework, there is indeed a very close correlation between dynamic frequency counts and simulator cycles. Before listing the correlation results we first describe a modification we made to the SimpleScalar cycle accurate simulator to make it faster.

Function	Inst	Blk	Brch	Loop	Fn_inst	sLen	CF	Leaf	% from opt.		within ? % of opt.		
									Batch	Worst	opt	2%	5%
AR_btbl_b...(b)	83	3	1	0	40	7	1	2	0.00	4.55	50.00	50.00	100.00
BW_btbl_b...(b)	68	3	1	0	56	7	1	4	0.00	4.00	50.00	50.00	100.00
bit_count(b)	36	9	5	1	155	11	5	4	1.40	1.40	50.00	100.00	100.00
bit_shifter(b)	47	10	7	1	147	8	9	3	0.00	3.96	66.67	66.67	100.00
bitcount(b)	133	3	1	0	86	8	1	10	2.40	4.33	10.00	10.00	100.00
main(b)	220	22	15	2	92834	21	91	171	8.33	233.31	3.51	14.62	14.62
ntbl_bitcnt(b)	43	6	3	0	253	9	2	20	18.69	18.69	10.00	10.00	10.00
ntbl_bitc...(b)	138	3	1	0	48	7	1	8	4.09	4.68	25.00	25.00	100.00
dequeue(d)	76	6	3	0	102	7	3	14	0.00	12.00	42.86	42.86	42.86
dijkstra(d)	354	30	22	3	86370	20	18	1168	0.04	51.12	0.34	23.63	29.11
enqueue(d)	124	15	10	1	570	12	6	9	0.20	4.49	22.22	66.67	100.00
main(d)	175	21	15	3	8566	18	28	143	4.29	75.32	2.80	4.20	51.05
print_path(d)	63	6	3	0	185	12	2	8	5.39	16.78	25.00	25.00	25.00
qcount(d)	12	3	1	0	7	3	1	1	0.00	0.00	100.00	100.00	100.00
CheckPoin...(f)	35	6	3	0	60	7	3	10	50.00	75.00	20.00	20.00	20.00
IsPowerOf...(f)	30	9	7	0	378	9	6	24	0.00	42.86	12.50	12.50	12.50
NumberOfB...(f)	59	11	7	1	2302	14	10	48	28.89	33.33	2.08	2.08	29.17
ReverseBits(f)	44	8	5	1	238	10	7	2	0.00	0.00	100.00	100.00	100.00
byte_reve...(h)	146	8	5	1	1661	13	11	24	1.82	42.44	4.17	33.33	41.67
main(h)	101	16	11	1	12168	18	153	241	7.14	100.00	0.00	0.00	0.00
sha_final(h)	155	7	4	0	1738	13	3	64	0.00	14.63	40.62	40.62	65.62
sha_init(h)	87	3	1	0	80	8	1	9	0.00	26.67	44.44	44.44	44.44
sha_print(h)	60	3	1	0	44	7	1	7	9.09	27.27	14.29	14.29	14.29
sha_stream(h)	55	8	5	1	251	13	4	10	9.15	9.15	0.00	80.00	80.00
sha_trans...(h)	541	33	25	6	343162	26	95	2964	6.03	103.35	0.00	14.98	59.01
sha_update(h)	118	11	7	1	5595	17	48	37	0.07	82.02	0.00	86.49	86.49
finish_in...(j)	5	3	1	0	3	2	1	1	0.00	0.00	100.00	100.00	100.00
get_raw_row(j)	60	6	3	0	84	9	1	8	0.00	7.69	87.50	87.50	87.50
jinit_rea...(j)	45	3	1	0	22	6	1	2	0.00	0.00	100.00	100.00	100.00
main(j)	465	40	28	1	33620	17	12	153	5.52	6.00	0.00	0.00	0.00
parse_swi...(j)	1228	198	144	1	200397	18	53	2365	6.67	64.85	0.34	1.69	3.55
pbm_getc(j)	41	11	7	1	73	9	8	4	0.00	22.13	50.00	50.00	50.00
read_pbm_...(j)	134	27	21	2	3174	13	19	42	6.70	69.27	4.76	14.29	16.67
select_fi...(j)	149	25	21	0	400	10	10	12	0.00	7.14	25.00	25.00	75.00
start_inp...(j)	795	63	50	1	7018	15	37	52	1.72	27.59	23.08	76.92	76.92
write_std...(j)	16	3	1	0	8	4	1	1	0.00	0.00	100.00	100.00	100.00
init_search(s)	103	13	9	2	1348	14	11	27	0.37	459.31	3.70	51.85	51.85
main(s)	175	19	12	3	30975	19	12	175	0.00	64.02	4.57	6.86	6.86
strsearch(s)	128	23	17	2	46545	16	86	1644	1.48	138.68	0.18	0.73	4.01
unexecuted(71)	168.6	16.2	11.7	0.9	26555.2	12.1	31.6	147.4	0.00	0.00	0.00	0.00	0.00
average	166.7	16.9	12.0	0.9	25362.6	12.0	27.5	182.8	4.60	47.64	30.68	42.02	56.08

Function - function name followed by benchmark indicator [(b)-bitcount, (d)-dijkstra, (f)-fft, (h)-sha, (j)-jpeg, (s)-stringsearch], Inst - number of instructions in unoptimized function, Blk - number of basic blocks, Brch - number of conditional and unconditional transfers of control, Loop - number of loops, Fn_inst - number of distinct control flow instances, sLen - largest active optimization phase sequence length, CF - number of distinct control flows, Num - Number of leaf function instances, % from opt - % performance difference between *Batch* and *Worst* leaf and Optimal, within ? % of optimal - what percentage of leaf function instances are within "?"% from optimal

Table 3. Dynamic Frequency Measures for Executed Functions for MiBench Benchmarks Used in the Experiments

6.1 Mixed Mode Simulator

The SimpleScalar simulator toolset [22] includes many different simulators intended for different tasks. Most useful for our purposes are the two simulators *sim-uop* and *sim-outorder*. *Sim-uop* is a functional simulator which implements the architecture, only performing the actual program execution. *Sim-outorder* is a performance simulator which implements the microarchitecture, modeling the system resources and internals in addition to executing the program. Thus, *sim-uop* is relatively fast but only provides dynamic instruction counts, whereas *sim-outorder* is able to provide processor cycles, but takes many times longer to run. In our experiments we use *sim-outorder* with the *inorder* flag for the ARM, which is generally an in-order processor for most implementations.

For our tests we only concentrate on one function at a time. Although the cache and global branch access patterns of the remaining functions in the application, as well as the library functions can affect the performance of the current function being optimized, the side effect should generally be small. In such a scenario it would be ideal if we could run only the current function through the cycle accurate simulator, and run all remaining functions using the faster functional simulator. This method has the potential of reducing the time required for simulations close to the level provided by the faster functional simulator, while still being able to provide accurate processor cycles for the function in question.

Sim-outorder already has a mode by which we can do only the functional simulation for some number of initial instructions before starting the cycle accurate simulation. But once the cycle simulation was started, it was not possible to go back to the functional mode. We extended *sim-outorder* to include the ability of going back to the functional mode from the cycle mode, so that we can essentially flip back and forth between the two modes whenever desired. Before each simulator run required during our experiments, we first use the Unix *nm* utility to get the start and end instruction addresses for the current function. Later during simulation, we

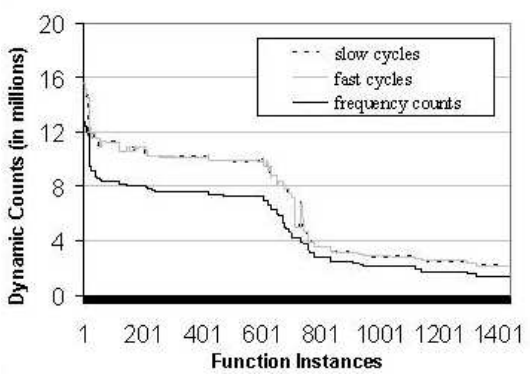

Figure 4. Correlation between Processor Cycles and Frequency Counts for *init_search*

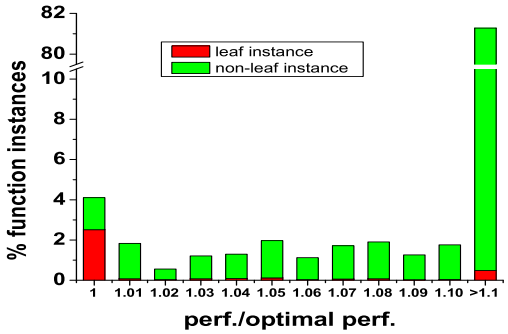

Figure 5. Average Distribution of Dynamic Frequency Counts

switch to the cycle mode only when the program counter is between this address range for the current function. Whenever the address falls out of this range we wait for the pipeline to empty and then revert back to functional simulation. This approach gives us substantial savings in time with very little loss in accuracy.

6.2 Complete Function Correlation

Even after using the mixed mode SimpleScalar simulator, it is very time consuming to simulate the application for all function instances in every function, instead of simulating the program only on encountering new control flows. We have simulated all instances of a single function completely to provide an illustration of the close correlation between processor cycles and our estimate of dynamic frequency counts. Figure 4 shows this correlation for all the function instances for the *init_search* function in the benchmark *stringsearch*. This function was chosen mainly because it is relatively small, but still has a sufficient number of distinct function instances to provide a good example.

In addition to comparing the dynamic frequency estimates and *fast* (mixed-mode) simulator cycles, Figure 4 also shows that the mixed-mode cycles are almost identical to the actual cycles from *sim-outorder*. This shows that our fast mode of using *sim-outorder* provides very accurate estimates. All these performance numbers are sorted on the basis of dynamic frequency counts. Thus, we can see that our estimate of dynamic frequency counts closely follows the processor cycles most of the time. What is more important is that the correlation gets better as the function is better optimized. The excellent correlation between dynamic frequency estimates and fast/slow cycles for the optimized function instances allows us to predict the function instances with good/optimal cycle counts with a high level of confidence.

6.3 Correlation for Leaf Function Instances

Figure 5 shows the distribution of the dynamic frequency counts as compared to the optimal, averaged over all function instances. From this figure we can see that the performance of the leaf function instances is typically very close to the optimal performance, and that leaf instances comprise a significant portion of optimal function instances as determined by the dynamic frequency counts. From the discussion in Section 5 we know that for more than 87% of the functions in our benchmark suite there was at least one leaf function instance that achieved optimal dynamic frequency counts. Moreover, it is important to note that the leaf instances constitute the only set of function instances that can be produced by the class of aggressive compilers that iteratively apply optimization phases until no additional improvements can be made. Since the leaf function instances achieve good performance across all our functions, it is worthwhile to concentrate on leaf function instances. We can see at the same time from Table 3 that most functions typically have very few leaf instances. This makes it possible to get simulator cycle counts for only the leaf function instances and compare these values to our dynamic frequency counts. In this section we show the correlation between dynamic frequency counts and simulator cycle counts for only the leaf function instances for all executed functions in our benchmark suite.

The correlation between dynamic frequency counts and processor cycles can be illustrated by various techniques. A common method of showing the relationships between variables (data sets) is by calculating Pearson's correlation coefficient for the two variables. The Pearson's correlation coefficient can be calculated by using the formula:

$$Pcorr = \frac{\sum xy - \frac{\sum x \sum y}{n}}{\sqrt{(\sum x^2 - \frac{(\sum x)^2}{n}) * (\sum y^2 - \frac{(\sum y)^2}{n})}} \quad (1)$$

In Equation 1 x and y correspond to the two variables, which in our case are the dynamic frequency counts and simulator cycles, respectively. Pearson's coefficient measures the strength and direction of a linear relationship between two variables. Positive values of *Pcorr* in Equation 1 indicate a relationship between x and y such that as values for x increase, values of y also increase. The closer the value of *Pcorr* is to 1, the stronger is the linear correlation between the two variables. Thus, $Pcorr = +1$ indicates *perfect* positive linear correlation between x and y.

It is also worthwhile to study how close the processor cycle count for the function instance that achieves the best dynamic measure, is to the best overall cycle count over all the leaf function instances. To calculate this measure, we first find the best performing function instance(s) for dynamic frequency counts and obtain the corresponding simulator cycle count for that instance. In cases where multiple function instances provide the same best dynamic frequency count, we obtain the cycle counts for each of these function instances and only keep the best cycle count amongst them. We then obtain the simulator cycle counts for all leaf function instances and find the best cycle count in this set. We then calculate the following ratio for each function:

$$Lcorr = \frac{best\ overall\ cycle\ count}{cycle\ count\ for\ best\ dynamic\ freq\ count} \quad (2)$$

The closer the value of Equation 2 comes to 1, the closer is our estimate of optimal by dynamic frequency counts to the optimal by simulator cycles.

Table 4 lists our correlation results for leaf function instances over all executed functions in our benchmarks. The column, labeled *Pcorr* provides the Pearson's correlation coefficient according to Equation 1. An average correlation coefficient value of 0.96 implies that there is excellent correspondence between dynamic frequency counts and cycles. The next column shows the value of *Lcorr* calculated by Equation 2. The following column gives the number of distinct leaf function instances which have the same best dynamic frequency counts. This number provides an estimate of how many function instances we would need to simulate to obtain optimal cycles, in case the cycles for any one of these function instances actually achieves overall optimal performance (only for leaf function instances in this case). Thus, it can be seen that on average the value of *Lcorr* is very close to 1, and we would need to simulate less than 5 function instances per function to obtain the leaf instance achieving the best cycle counts. The next two columns show the same measure of *Lcorr* by Equation 2, but instead of considering only the best leaf instances for dynamic frequency counts, they consider all leaf instances which come within 1% of the best dynamic frequency estimate. Considering more instances increases the number of simulator runs we need to perform to get best cycles, but also allows our estimated good leaf instances to get even closer to being able to predict optimal simulator cycles.

The conclusions of this study are limited since we only considered leaf function instances. It would not be feasible to get cycle counts for all function instances over all functions. In spite of this restriction, the results are interesting and noteworthy since they show that a combination of static and dynamic estimates of performance can predict pure dynamic performance with a high degree of accuracy. This result also leads to the observation that we should typically only need to simulate a very small percentage of *good* function instances as indicated by dynamic frequency counts to obtain the optimal function instance by simulator cycles.

Function	Pcorr	Lcorr 0%		Lcorr 1%	
		Diff	nLf	Diff	nLf
AR_btbl_b...	1.00	1.00	1	1.00	1
BW_btbl_b...	1.00	1.00	2	1.00	2
bit_count	1.00	1.00	2	1.00	2
bit_shifter	1.00	1.00	2	1.00	2
bitcount	0.89	0.92	1	0.92	1
main	1.00	1.00	6	1.00	23
ntbl_bitc...	0.99	0.95	2	0.95	2
ntbl_bitcnt	1.00	1.00	2	1.00	2
dequeue	0.99	1.00	6	1.00	6
dijkstra	1.00	0.97	4	1.00	269
enqueue	1.00	1.00	2	1.00	4
main	0.98	1.00	4	1.00	4
print_path	1.00	1.00	2	1.00	2
qcount	1.00	1.00	1	1.00	1
CheckPoin...	0.95	1.00	2	1.00	5
IsPowerOf...	0.93	0.98	3	1.00	24
NumberOfB...	0.84	1.00	1	1.00	20
ReverseBits	1.00	1.00	2	1.00	2
byte_reve...	0.89	1.00	1	1.00	3
main	0.71	1.00	25	1.00	74
sha_final	0.72	0.82	26	1.00	50
sha_init	0.98	1.00	4	1.00	9
sha_print	0.95	0.88	1	1.00	6
sha_stream	1.00	1.00	1	1.00	8
sha_trans...	0.97	1.00	2	1.00	35
sha_update	0.98	1.00	14	1.00	32
finish_in...	1.00	1.00	1	1.00	1
get_raw_row	1.00	1.00	7	1.00	7
jinit_rea...	1.00	1.00	2	1.00	2
main	1.00	0.99	2	1.00	153
parse_swi...	0.95	1.00	8	1.00	16
pbm_getc	0.99	1.00	2	1.00	2
read_pbm_...	0.73	0.98	2	0.98	2
select_fi...	0.97	0.90	3	1.00	12
start_inp...	0.95	0.99	12	0.99	15
write_std...	1.00	1.00	1	1.00	1
init_search	1.00	1.00	1	1.00	14
main	1.00	1.00	8	1.00	12
strsearch	1.00	1.00	3	1.00	3
average	0.96	0.98	4.38	0.996	21

Pcorr - Pearson's correlation coefficient, Lcorr - ratio of cycles for dynamic frequency to best overall cycles (0% - optimal, 1% - within 1 percent of optimal frequency counts), Diff - ratio for *Lcorr*, nLf - number of leaves achieving the specified dynamic performance

Table 4. Correlation Between Dynamic Frequency Counts and Simulator Cycles for Leaf Function Instances

7. Genetic Algorithm Performance Results

Since it was generally considered infeasible to enumerate the optimization phase order search space, many researchers employ heuristic algorithms, such as grid-based searches, hill-climbing algorithms etc. to adaptively search a subset of the complete optimization phase order space [4, 5, 8, 13]. Such heuristic algorithms are believed, to provide better solutions faster than just a random exploration of the search space. However, till this date there has not been any comprehensive work to study how good the solutions provided by such heuristic algorithms compare with the best possible solution since researchers always considered the optimization phase order space too large to fully evaluate. Since we now know the optimal function instance for each function, it is worthwhile to perform a case-study to compare the results of our genetic algorithm to investigate how close it comes to the optimal dynamic frequency counts solution and how long it takes to achieve this optimum for each function. Further, we attempt to exploit some of the information about the optimization interactions collected during our exploration of the optimization phase order space to determine their effect on the performance of the genetic algorithm.

The genetic algorithm setup used for these experiments is similar to some earlier experimental setups used by other researchers to assess the merit of using genetic algorithms to address the phase ordering problem [4, 5]. Chromosomes in the genetic algorithm correspond to optimization sequences, and *genes* in the chromosome correspond to optimization phases. The set of chromosomes currently under consideration constitutes a *population*. The number of *generations* indicates how many sets of populations are to be evaluated. As in earlier experiments [4, 5], we have 20 *chromosomes* per generation. However, in our algorithm the number of generations is kept floating, as opposed to the fixed number of generations (100) used earlier. The algorithm now terminates after the generation in which it finds the optimal dynamic frequency counts solution, or if the best solution found by the genetic algorithm does not improve over its current best for more than a 100 generations. The potential increase in the number of generations that could result from the change in the terminating condition can be easily sustained since now the application only needs to be simulated for new control flows, as explained in Section 5. The sequence length for use by the genetic algorithm is determined by multiplying the largest *active* sequence length found for that function (see Table 3) by 1.25. The multiplication factor is to take into account *dormant* phases since the chromosomes applied during the genetic algorithm represent the *attempted* sequence and not the *active* sequence. To produce the next generation we first sort the chromosomes in the current generation in ascending order of their dynamic frequency counts. During *crossover* we replace four chromosomes from the lower half of the sorted list by repeatedly selecting two chromosomes from the upper half of the list and replacing the lower half

of the first chromosome with the upper half of the second chromosome and vice-versa to produce two new chromosomes each time. During *mutation* we replace a phase with another random phase with a probability of 5% for chromosomes in the upper half of the population and 10% for the chromosomes in the lower half.

We also performed a second set of experiments with the genetic algorithm by changing the *mutation* routine to take into account optimization phase interaction results. The complete enumeration of the phase order space provides us with useful information about how often some phase *enables* or *disables* some other phase. This information was used in our previous study to obtain faster batch compilation [1]. For the current study, instead of selecting a purely random replacement phase during *mutation*, we use this information to select the phase based on the probability of it being active at that point. The current phase probabilities are modified after every phase by using the algorithm below.

```
/* opt is the last phase performed and becomes dormant */
prob[opt] = 0.0
/* determine probability of each phase being active */
for(i=0 ; i < seq_length ; i++)
     prob[i] = (1.0-prob[i])*enable[opt][i] +
                (prob[i])*disable[opt][i]
```

Enable and *disable* are the phase probabilities for each phase interaction with all other phases. The enabling/disabling probabilities for each function were determined by averaging these probabilities over all the functions, except for the current function whose optimization phase order is being searched by the genetic algorithm. The initial probabilities for each phase reflects the percentage of time it was found to be active at the start of the optimization process (i.e. for the unoptimized function instance). The technique of only simulating the program on encountering a yet unseen controlflow gives us a huge time savings beyond our earlier approaches for making genetic algorithms faster [8] (for example, from 173 seconds to 18 seconds for *init_search* in *stringsearch*).

Table 5 shows the results of these experiments. The three columns under *Original GA* list the results using unbiased random mutation, and the next four columns under *Modified GA* illustrate the results with the mutation phase changed to reflect phase interaction probabilities. From these results it can be seen that genetic algorithms typically take only a few generations to reach their best result. The original genetic algorithm was occasionally not able to find the optimal instance under the test conditions, which resulted in a performance loss of 0.51% on average. After accounting for the phase interactions during *mutation*, the modified genetic algorithm was able to find the optimal instance for all but one of the executed functions in our benchmark suite. This shows that using phase probabilities helps bias the genetic algorithm searches towards better sequences faster. It is possible that the original genetic algorithm left to run for a greater number of generations will eventually reach the optimal solution as well. It should however be noted that the increase in the number of generations has only been made possible due to the ability to accurately estimate the frequency counts without actually executing the application in most cases. This study shows that adaptive algorithms work very well in our test suite and often are able to find the optimal phase ordering for the compiler we used. The reason for this is quite evident from Table 3 which shows that most functions typically have many minima, as well as many other points close to the optimal solution.

Function	Original GA			Modified GA			
	Gen	Opt	Df	Gn1	Gn2	Opt	Df
AR_btbl_b...	1	Y	0.00	1	1	Y	0.00
BW_btbl_b...	1	Y	0.00	1	1	Y	0.00
bit_count	25	Y	0.00	25	25	Y	0.00
bit_shifter	19	Y	0.00	11	11	Y	0.00
bitcount	4	Y	0.00	4	4	Y	0.00
main	58	Y	0.00	23	23	Y	0.00
ntbl_bitcnt	21	N	6.55	7	44	Y	0.00
ntbl_bitc...	6	Y	0.00	4	4	Y	0.00
dequeue	32	Y	0.00	9	9	Y	0.00
dijkstra	1	Y	0.00	1	1	Y	0.00
enqueue	38	Y	0.00	17	17	Y	0.00
main	28	N	3.96	14	123	Y	0.00
print_path	4	Y	0.00	3	3	Y	0.00
qcount	1	Y	0.00	1	1	Y	0.00
CheckPoin...	1	Y	0.00	1	1	Y	0.00
IsPowerOf...	9	Y	0.00	15	15	Y	0.00
NumberOf...	49	Y	0.00	94	94	Y	0.00
ReverseBits	17	Y	0.00	20	20	Y	0.00
byte_reve...	14	N	4.69	17	113	Y	0.00
main	18	Y	0.00	14	14	Y	0.00
sha_final	1	Y	0.00	1	1	Y	0.00
sha_init	9	Y	0.00	12	12	Y	0.00
sha_print	14	Y	0.00	4	4	Y	0.00
sha_stream	84	Y	0.00	37	37	Y	0.00
sha_trans...	25	N	0.73	226	227	N	0.70
sha_update	10	Y	0.00	14	14	Y	0.00
finish_in...	1	Y	0.00	1	1	Y	0.00
get_raw_row	2	Y	0.00	2	2	Y	0.00
jinit_rea...	22	Y	0.00	14	14	Y	0.00
main	71	Y	0.00	20	20	Y	0.00
parse_swi...	14	N	4.07	13	26	Y	0.00
pbm_getc	31	Y	0.00	41	41	Y	0.00
read_pbm_...	16	Y	0.00	12	12	Y	0.00
select_fi...	34	Y	0.00	10	10	Y	0.00
start_inp...	11	Y	0.00	4	4	Y	0.00
write_std...	1	Y	0.00	1	1	Y	0.00
init_search	5	Y	0.00	4	4	Y	0.00
main	16	Y	0.00	3	3	Y	0.00
strsearch	17	Y	0.00	13	13	Y	0.00
average	18.7	0.87	0.51	18.3	24.9	0.97	0.02

Gen - generation to reach the best solution, Opt - whether optimal was found for this function, Df - how worse the best GA solution was from optimal, Gn1 - generations required by the modified GA to reach a solution that is at least as good as that found by the Original GA, Gn2 - generation to reach best solution for modified GA.

Table 5. Genetic Algorithm Results

8. Future Work

In the future, we plan to investigate making a variety of improvements to the current work. We will attempt improvements to make the optimization phase order enumeration algorithm more efficient. Currently, for each distinct function instance at the current level, the enumeration algorithm applies all optimization phases at the next level to check if any of them lead to a new function instance. In practice, most of these phases turn out to be dormant. We can use the enabling/disabling and independence phase interaction relationships gathered from previous enumeration runs to avoid optimization sequences which are predicted to not have a sufficient probability of success. It may also be possible to use the phase interaction relationships to determine phases which can then be grouped together to reduce the number of distinct optimizations during phase order enumeration. Secondly, we plan to improve non-exhaustive searches of the phase order space. Presently the only feedback we get from each optimization phase was whether it was active or dormant. We do not keep track of the number and type of actual changes for which each phase is responsible. Keeping track of this information would be very useful to get more accurate phase interaction information to avoid dataflow analysis that is unaffected or to better choose the next phase to apply.

9. Conclusions

The ultimate goal of the exhaustive enumeration of the optimization phase order search space for a function is to find an instance that will achieve optimal dynamic execution performance. In this paper we have shown that for most of the functions in our benchmark suite, it is possible to obtain the optimal function instance w.r.t. our measure of dynamic frequency counts, in a reasonable amount of time. We do this by limiting the number of simulations during the exhaustive enumeration of the optimization phase order search space to only those function instances which have a control flow that was not previously encountered. For all remaining function instances for that control flow we estimate the dynamic frequency measures from the execution counts of the basic blocks. We further show that our measure of dynamic frequency counts correlates very well with actual simulator cycles when the frequency counts compare versions of the program that have the same control flow. This leads us to the conclusion that if we could determine a small number of *good* function instances quickly (from the exhaustive set of all possible function instances) using our measure of dynamic frequencies, then it is possible to simulate only the function instances in this set. We found that the function instance achieving the best simulator cycles from this smaller set will likely be optimal, or very close to optimal for the possible phase orderings in the compiler we used. Finally, we showed that even basic genetic algorithms are often able to find the optimal function instance (for dynamic frequency counts) for our compiler in a small number of generations, and that their performance can still be improved by exploiting phase interaction information during the algorithm.

10. Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants EIA-0072043, CCR-0208892, CCR-0312493, CCF-0444207, and CNS-0305144.

References

[1] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson. Exhaustive optimization phase order space exploration. In *Proceedings of the Fourth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, March 26-29 2006.

[2] Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the 15th annual workshop on Microprogramming*, pages 125–133. IEEE Press, 1982.

[3] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming*, pages 137–146. ACM Press, 1990.

[4] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, May 1999.

[5] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, pages 12–23. ACM Press, 2003.

[6] T. Kisuki, P. Knijnenburg, , and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. PACT*, pages 237–246, 2000.

[7] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code Generation and Optimization*, pages 204–215. IEEE Computer Society, 2003.

[8] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, June 2004.

[9] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, 1997.

[10] Min Zhao, Bruce R. Childers, and Mary Lou Soffa. A model-based framework: An approach for profit-driven optimization. In *Proceedings of the international symposium on Code generation and optimization*, pages 317–327, Washington, DC, USA, 2005.

[11] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, New York, NY, USA, 2004. ACM Press.

[12] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, , and E. Rohou. Iterative compilation in a non-linear optimisation space. Proc. Workshop on Profile and Feedback Directed Compilation.Organized in conjuction with PACT'98, 1998.

[13] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim.*, 2(2):165–198, 2005.

[14] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, F. Bodin, , and H.A.G. Wijshoff. A feasibility study in iterative compilation. In *Proc. ISHPC'99, volume 1615 of Lecture Notes in Computer Science*, pages 121–132, 1999.

[15] Elana D. Granston and Anne Holler. Automatic recommendation of compiler options. 4th Workshop of Feedback-Directed and Dynamic Optimization, December 2001.

[16] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizatons. Proc. 2nd Workshop on Feedback Directed Optimization, 1999.

[17] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Generating new general compiler optimization settings. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 161–168, New York, NY, USA, 2005. ACM Press.

[18] P.M.W. Knijnenburg, T. Kisuki, K. Gallivan, and M.F.P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. In *Proc. FDDO-3*, pages 31–40, 2000.

[19] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. *SIGPLAN Not.*, 29(6):85–96, 1994.

[20] K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: Adaptive compilation made efficient. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 69–78, June 15-17 2005.

[21] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*, pages 329–338. ACM Press, 1988.

[22] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[23] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[24] W. Peterson and D. Brown. Cyclic codes for error detection. In *Proceedings of the IRE*, volume 49, pages 228–235, January 1961.

[25] Jack W. Davidson and David B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, November 1991.