

Whole-Program Optimization of Global Variable Layout

Nathaniel McIntosh, Sandya Mannarswamy, Robert Hundt
Java, Compilers, and Tools Laboratory
Hewlett-Packard Company

nathaniel.mcintosh@hp.com, sandya.s.mannarswamy@hp.com,
robert.hundt@hp.com

ABSTRACT

On machines with high-performance processors, the memory system continues to be a performance bottleneck. Compilers insert prefetch operations and reorder data accesses to improve locality, but increasingly seek to modify an application's data layout to reduce cache miss and page fault penalties. In this paper we discuss *Global Variable Layout* (GVL), an optimization of the placement of entire static global data objects in the binary. We describe two practical methods for GVL in the HP-UX Integrity optimizing compiler for the Itanium $\text{\textcircled{C}}$ architecture. The first layout strategy relies on profile feedback, collaboratively employing the compiler, the linker and a pre-link tool to facilitate reordering. The second strategy uses whole-program analysis to drive data layout decisions, and does not require the use of a dynamic profile. We give a detailed description of our implementation and evaluate its performance for the SPEC integer benchmark programs, as well as for a large commercial database application.

Categories and Subject Descriptors

D.3.4.1 [Software]: PROGRAMMING LANGUAGES, Processors—*Compilers, Optimization*

General Terms

Languages, Performance

Keywords

compiler-directed memory management, data caches, global variable layout

1. INTRODUCTION

On machines with high performance processors, the memory subsystem continues to be a major performance bottleneck. It is not uncommon for large commercial applications to spend 50 to 70 percent of their execution time

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PACT'06, September 16–20, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-264-X/06/0009 ...\$5.00.

stalled on data cache accesses [6]. Compilers therefore try to improve an application's cache locality and reuse with a wide range of techniques. For scientific programs with regular loop patterns, traditional loop nest optimizations have proven to be effective [16, 1]. For irregular programs with complex data structures and references, compilers are trying to modify data structures and, for example, reorder structure fields, change field types, or break individual structures into pieces [15, 3, 4, 27, 24, 30, 22, 11]. Compilers use sophisticated algorithms to place prefetch instructions into the code stream to further hide cache and memory latencies [20]. Similar techniques are employed by dynamic compilers, which additionally employ the garbage collector to improve locality by collocating heap objects [23, 7, 10, 5]. A related technique has been proposed to collocate objects in pools on the heap for C/C++ programs [17].

In this paper, we focus on “global variable layout”, a transformation wherein the compiler seeks to optimize the placement of objects in the application's entire static data segment. The order of items within the data segment has an effect on how well a program makes use of the cache memory in the machine on which it is executing. If a frequently-accessed data item F is placed adjacent to an infrequently accessed data item Q , then when the processor issues a load of F , it may wind up bringing in Q as well (if they share the same cache line). This is wasteful in terms of the cache footprint of the program. If “hot” (frequently accessed) data items are co-located, this can reduce the amount of “useless” data brought into the cache, resulting in fewer cache misses overall. Figure 1 shows an example. If the two “hot” data items A and B are positioned next to each other, we only need to bring a single line into the cache to access both of them in sequence.

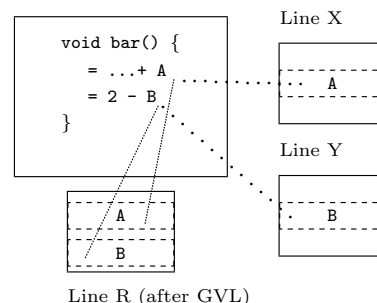


Figure 1: Co-locating data items

For multiprocessor systems with hardware cache coherence, the layout of global objects can have an impact on coherence traffic. If a heavily-written global scalar W is placed into the same cache line as an infrequently written but heavily read scalar R , the two access patterns can combine to produce useless coherence traffic. Writes to W will invalidate remote cache lines containing R , even though the value of R has not actually been altered. Factoring these effects into the selection of data layout for multiprocessor applications has the potential to reduce coherence traffic and improve overall cache hit ratios.

In this paper we describe two strategies for compiler-selected global data layout, we examine their performance, and we evaluate their relative strengths and weaknesses. The first approach uses dynamic profile data feedback, ordering global data items using heuristics based on an execution profile. In this approach, the actual reordering of data items is performed by the linker, much in the same way that the linker reorders procedures for improved instruction cache performance. The second approach does not require the use of an execution profile, but instead uses whole-program (interprocedural) analysis and optimization to drive the layout heuristics. In this second approach, the interprocedural optimizer performs the actual re-arranging of the data, without linker involvement. The two strategies described here are both implemented in a production compiler/linker tool chain for the Itanium architecture. We have applied the techniques to improve performance for the SPEC benchmarks and for a commercial database server application.

Earlier work [2, 8] has discussed data layout for global variables. These prior approaches, however, have required the use of a separate binary rewriting tool or specialized memory profiling tool to generate data to drive the layout (typically requiring simulation, which is impractical for most large applications). There are many areas in which our method is significantly different from earlier work, including the following:

- exploiting interprocedural analysis to enable data layout for applications where execution profiles are unavailable.
- partitioning of variables by size, storage class, and initialization class. This facilitates layout of read-only data items, without losing any savings reaped by placing read-only items in the text segment.
- packing of variables based on compiler-derived alignment information, so as to reduce any inter-variable padding (as opposed to post-link-time approaches, which have to be conservative with respect to alignment).
- differentiating between packing short and long variables; packing short variables by cache lines, which insures that benefits of GVL are actually realized.

This paper is organized as follows. In Section 2, we briefly outline our compiler and linker framework, which provides the software infrastructure underlying our implementation. In Sections 3 and 4, we describe the data layout implementation in detail, including the roles played by the compiler and linker. In Section 5, we present experimental results for certain SPEC benchmarks for both these approaches. In Section 6, we present our experience with applying the

dynamic-profile version of our techniques to a large commercial application, the Oracle database server. We discuss prior work in Section 7, and in Section 8, we outline ideas for future extensions to this work in the area of data layout for parallel applications. Finally, we summarize our conclusions in Section 9.

2. FRAMEWORK

The techniques described in this paper are incorporated into the HP-UX Integrity © compiler/linker tool chain, which includes support for both profile-based optimization and for interprocedural analysis and optimization. Here we give an overview of these features, since they provide the platforms upon which our reordering techniques are based.

2.1 Interprocedural analysis and optimization

Interprocedural optimization (IPO) is performed in our compiler by the SYZYGY high-level optimizer component [19]; it is enabled via the “-ipo” compiler command line option, or when the user requests the highest level of optimization (level 4). IPO is implemented in three logical phases. First, there is the SYZYGY front end (FE) phase, in which the compiler analyzes the source file and produces an object file containing high level IELF intermediate code, annotated with summary information for the different analyses. Each source file is analyzed independently, which means that many instances of the FE can be run in parallel.

The second phase is the monolithic interprocedural optimization phase (IPO), in which the interprocedural optimizer (invoked by the linker when IELF files are detected) performs whole-program analysis and transformations. The optimizer reads the IELF files that contribute to the link, analyzes them, carries out interprocedural transformations, and emits modified IELF files as output.

The third phase, the SYZYGY back end (BE), reads in a modified IELF file and performs routine-level optimizations, finishing with a call to the low-level optimizer to complete code generation. The BE phase is also parallelizable; by default, the IPO phase emits a makefile into the temporary directory containing the modified IELF files, and then invokes a parallel `make` utility to carry out the process of code generation. The resulting ELF objects are then linked normally to form the final load module. Additional details can be found in [19].

2.2 Profile-based optimization

The HP-UX Integrity compiler uses a two-pass methodology for profile-based optimization [21, 18].

In the first pass, the user builds the target application with the “+Oprofile=collect” command line option, causing the compiler to insert profiling or instrumentation code into each function. The user then runs the instrumented program on a representative input to produce an execution profile; this profile includes counts for all basic blocks and edges in each function’s control flow graph, and also records calls made from each profiled function.

In the second pass, the user performs another build of the application in which the file containing the execution profile is fed back into the compiler for use during optimization, using the “+Oprofile=use” command line option. Profile feedback is supported for optimization level 2 and above; variable layout is enabled by default when profile feedback is in use.

2.3 Synthetic execution profiles

For programs compiled without feedback, the compiler internally creates a synthetic execution profile through the use of static heuristics [29]. This so-called “static” profile plays much the same role that dynamic feedback data does in terms of guiding optimizations.

The compiler first produces a synthetic profile for each procedure independently, taking into the control flow and characteristics of the function to make guesses about conditional branch behavior. In cases where interprocedural analysis is enabled, the interprocedural optimizer performs additional analysis, reconciling the independent profiles to produce a consistent, program-wide profile for the application.

3. DYNAMIC-PROFILE GVL

This section describes the first of our two GVL variants, profile-driven GVL. For applications compiled with dynamic profile feedback, our compiler performs global variable layout in three phases, with roles played by both the compiler and linker. The phases are:

- identify and characterize candidate variables
- select actual ordering
- perform reordering

The first phase is carried out by the compiler; the second phase takes place in an auxiliary tool invoked just prior to linking, and the last phase is performed by the linker itself.

3.1 Collecting variable profiles

The compiler builds a profile-annotated control flow graph (CFG) for each procedure, using the block and edge frequencies derived from the execution profile. No additional requirements are imposed on profile collection to drive global variable layout; the profile pertains only to the program’s control flow behavior. The compiler visits each block in the CFG, collecting the set of variables accessed in each block. For each variable, it records the variable name, storage class, size, and alignment requirement in the table, along with read and write access counts from the block execution frequency. Figure 2 shows an example.

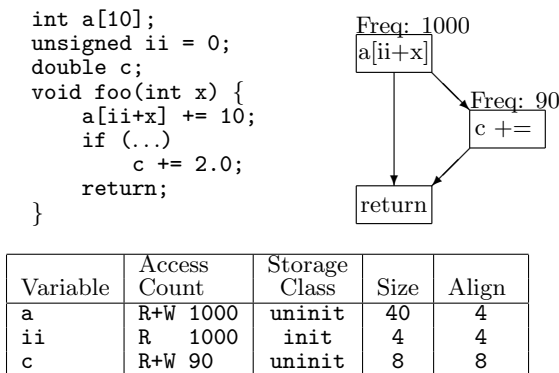


Figure 2: Candidate identification example

The compiler also collects affinity information during this phase. We say that two variables X and Y have “temporal affinity” if accesses to the two variables are likely to take place close together in time (hence making them good

candidates for co-location). Accurately capturing the exact affinity levels between all variables in a program would be very expensive, which means that we must rely on heuristics. Compilers can derive a limited amount of affinity information by looking for cases where variables are accessed within the same basic block. It is difficult for a compiler to get global, application-wide picture of affinity, however, without taking into account effects between procedures, such as when two variable accesses are separated by a procedure call. To address these problems, we rely on interprocedural analysis in the static profile case and on the dynamic call graph in the profile-driven case.

During the analysis phase, the compiler maintains a table that maps each “hot” variable V to a list of tuples of the form $\langle F, C \rangle$, where F is a function and C is the accumulated access count for variable V in function F . This table of variables is stored in the module’s object file, to be consumed during the linking phase.

3.2 Variable packaging

Several of the critical applications targeted for this work also require support for relinking the application at a customer site (for example, the HP-UX operating system kernel). For these applications, there is a very strong incentive to keep the link step as simple, fast, and reliable as possible, and to push as much of the optimization-related activity into phases prior to the final link step.

In order to meet this requirement, the compiler places each data item that is a candidate for reordering into a separate ELF section within its containing object file [28]. This is a key element of our framework; it drastically reduces the amount of work required by the linker, making it especially important for very large applications.

3.3 Memory model issues

One of the challenges we face with variable reordering is that a data item can be placed in one of many different sections/segments in the final load module, depending on the item’s size and characteristics, and depending on the runtime architecture rules. Figure 3 illustrates some of the various data sections/areas present in the case of the HP-UX Itanium runtime architecture [13]:

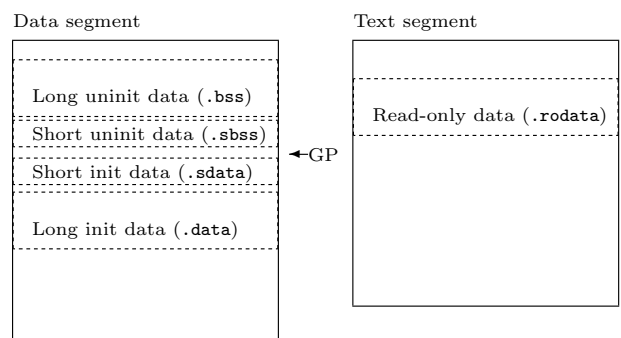


Figure 3: ELF runtime memory model

The compiler places short (8 bytes or less) data items in the `.sbss` and `.sdata` sections, and materializes addresses for these items directly off the global pointer (“GP”). The compiler places longer data items into the `.bss` and `.data`

sections, and materializes addresses for them by loading the address value from a linkage table entry for the item created by the linker. The short/long split and the size of the short data section is mandated in part by the Itanium instruction set, particularly the size of the constant offset used in common address computations. If a variable is declared as “const”, then the compiler will place it in the program’s `.rodata` section, which is logically part of the text segment. All of this partitioning poses problems for reordering, since two variables can only be co-located by the linker if they are in the same type of section.

It is also common for data items to be declared and used inconsistently. The following example illustrates:

```

a.c                                b.c
extern int *x;                          int x[20];
int q;                                   char *q = 0;
<accesses to x, q>                      ...

```

In this case the candidate variables are declared differently in different link units; this results in potential conflicts that have to be sorted out at link time during the layout phase. In the example above, file `a.c` contains a so-called “weak” or tentative definition of `q`, whereas in `b.c`, the variable `q` has a “strong” definition, due to the initialization. C language rules require that we favor the strong definition of `q` over the weak definition; this has to be taken into account by GVL.

3.4 Deriving the layout

The third phase, determination of the final variable layout, takes place in the initial stages of the linking step. After making an examination of its inputs, the linker invokes an auxiliary tool named “FDP”, which performs both procedure and data layout.

FDP starts by reading in the execution profile for the program, since in order to reorder procedures it has to construct a call graph [9]. In the call graph, nodes correspond to procedures, and edges between nodes correspond to dynamic call counts from the profile. FDP next reads in the summary sections previously generated by the compiler during candidate variable identification.

FDP builds a global table of candidate variables, resolving any conflicts relating to size, alignment, and storage class. It then separates the candidate variables into one of five partitions: short read-only, long read-only, short writable, long writable uninitialized, and long writable initialized. FDP performs layout for all short writable candidates together, since it can request that the linker promote short uninitialized candidates (`.sbss`) to short initialized (`.sdata`). This promotion does not require additional analysis: although we are increasing the size of the `.sdata` section, we are correspondingly decreasing the size of the `.sbss` section. If the linker were to instead promote items from `.data` to `.sdata`, for example, we would run the risk of creating an “unaddressable” short data item, causing a failure to link.

Next, FDP constructs a program-level *variable affinity graph* or VAG. This is an undirected graph where nodes correspond to variables, and edges carry a weight that indicates pseudo-temporal affinity. For each function `F`, we add a weighted edge between the nodes for variables `X` and `Y` if both `X` and `Y` are accessed in `F`. Using the previously constructed call graph, for each call edge $C \rightarrow D$, the affin-

ity graph constructor adds affinities between the variables accessed in `C` to the variables accessed in `D` (we use smaller affinity value here than in the previous step). Note that we need only construct a variable affinity graph for the variables within a given partition; we save time by constructing several small variable affinity graphs instead of a single large graph.

Long data item layout

FDP performs layout of long data items first; it uses a simple greedy algorithm for each of the three long partitions. The algorithm starts by selecting the “hottest” data item (highest access count) and then selects successor items based on total access count and on the affinity level (from the VAG) with the previous candidate.

Short data item layout

The layout algorithm for short data items is cache-line centric. The algorithm operates by creating a “virtual cache line”, that is, a logical container the size of a cache line. The layout algorithm then selects items from the current partition by access count and affinity until the current cache line is “full”, that is, until it has data items whose collected size adds up to the size of the line. It then arranges the items within the cache line so as to eliminate the need for any padding (the compiler has recorded size and alignment information for this purpose), to maximize affinity between adjacent variables, and to pack as many items into the line as possible. Figure 4 illustrates the layout selected for the hottest cache line in the SPECint2000 program `186.crafty`.

Variable	size (bytes)
<code>file_mask</code>	64
<code>pawn_score</code>	20
<code>largest_positional_score</code>	4
<code>mask_A7H7</code>	8
<code>all_pawns</code>	8
<code>root_total_black_pieces</code>	4
<code>root_total_white_pieces</code>	4
<code>nodes_searched</code>	4
<code>evaluations</code>	4
<code>pawn_probes</code>	4
<code>pawn_hits</code>	4

Figure 4: Cache line layout in `186.crafty`

Our experience was that reordering achieved slightly better results when targeting the 128-byte Itanium 2 second level cache line, as opposed to the 64-byte first level cache line.

Layout finalization

When the final layout is determined (for both procedures and data), FDP emits an ASCII file containing layout directives that is then consumed by the linker. This file contains the names of procedures and data items, along with alignment directives of the form “% align N” to insure that variables are positioned properly in relation to the start of the cache line. The linker enforces alignment by adding padding to the section in question.

In cases where ISVs want to support customers who re-link their applications on-site, the ASCII layout file generated by FDP can be captured and then shipped by the ISV

to the customer. The customer can then relink the application quickly, without any additional work to determine layout from the profile (and without a need to ship the file containing the execution profile).

3.5 Final reordering

The final linking stage is comparatively simple. The linker reads the layout file generated by FDP and reorders the associated ELF sections to achieve the desired layout, while inserting padding (if needed) to maintain the alignment specified by FDP.

4. SYNTHETIC-PROFILE GVL

Synthetic profile layout of global variables in our compiler is similar to the dynamic profile version of GVL described in the previous section. Synthetic-profile GVL employs heuristics based on interprocedural analysis instead of a dynamic profile to drive the process, however.

Synthetic-profile GVL is implemented as an interprocedural optimization within the SYZYGY high-level optimizer, as described in section 2.1. The phases or sub-tasks for this optimization are:

- Selection of candidates for variable layout and collection of profile counts for selected candidates on a routine basis. This happens in the SYZYGY FE phase.
- Aggregation of this information at whole program level, and based on the information, reordering of the layout of the selected candidates for the whole program as part of the interprocedural optimization. This happens in the SYZYGY IPO phase.

Candidate selection is done on a routine level basis, by traversing the synthetic-profile annotated control flow graph (CFG) of the routine for each basic block and recording the candidate information such as variable name, storage class, size, and alignment requirement in the GVL candidate table, along with read and write access counts from the block execution frequency. The GVL candidate table also contains the affinity information for variables. The GVL candidate table is added as annotation to the IELF file, which is written out at the end of FE phase.

In the IPO phase, we construct a whole-program GVL variable table, entering each candidate variable into the table as we examine each routine in the input IELF files. The algorithm then uses the strategies described in Section 3 to determine the final layout: separating the variables into partitions, constructing the variable affinity graph, and laying out the candidates.

The IPO phase selects a single link unit which will act as a host for all of the candidate variables to be reordered (the default is to choose the first link unit seen by the interprocedural optimizer). The optimizer then imports the variables from their home link units into the target link unit, along with any constants and type information associated with the variable; this process is similar to that used for promoting static variables to global variables during cross-module inlining. The original definitions of the variables are then deleted. Finally, the variables are placed in the desired order; the actual storage allocation is done in the BE phase of the compiler. The IPO communicates the ordering to the BE allocator via an IR annotation, and the BE storage allocator carries out the specified layout directives.

4.1 Floating point data items

The Itanium 2 micro-architecture is designed to keep floating point data items in the L2 cache, not the L1 cache. Floating point load and store instructions bypass the L1 cache and go directly to L2. This has implications for data layout; if a floating point data item F (accessed via `ldf/stf` instructions) is co-located with an integer data item I , a floating store to F will cause the L1 line containing I to be invalidated, resulting in an L1 cache miss the next time I is accessed. Synthetic-profile GVL addresses this issue by separating out floating point data from integer data during short writable layout. Though we did not see any specific degradations in SPEC benchmarks by not separating out the float data, this can be an issue in real life applications.

5. EXPERIMENTAL EVALUATION: SPEC

In this section, we present experimental results for global variable layout for SPEC2000 integer benchmarks, along with a selection of important public-domain integer applications. The baseline for our SPEC experiments for the dynamic profile driven GVL is our dynamic profile driven SPEC base configuration with all high level interprocedural optimizations enabled except for global variable layout. We used version A.06.11 (beta) of the HP ANSI C/C++ compiler. Results were obtained on a Hewlett-Packard Integrity rx2600 server with a 1500 MHz Intel Itanium processor, 6 GB of memory and 6 MB of L2 cache. Each benchmark was run three times for each configuration (GVL vs base), and the best time was selected. For identical binaries, run-to-run noise (worst time minus best time over best time) averaged 0.1% for SPECint2000, with a maximum value of 0.9% for 181.mcf. For the public-domain applications, run-to-run noise averaged 0.2%, with a maximum value of 1.0% for xalancbmk.

Benchmark	GVL entries	% improvement from GVL	
		dynamic	synthetic
SPECint2000			
164.gzip	107	-0.22	0.24
175.vpr	168	0.00	0.32
176.gcc	1412	1.29	0.00
181.mcf	15	0.16	0.12
186.crafty	432	6.08	3.94
197.parser	128	0.77	0.11
252.eon	1576	0.16	0.00
253.perlbmk	498	0.78	-0.48
254.gap	312	0.75	0.99
255.vortex	597	2.65	3.66
256.bzip2	62	0.19	0.19
300.twolf	316	-0.67	0.15
Public-domain applications			
perlbench	664	0.51	0.23
bzip2	35	-0.03	0.00
gcc	1368	0.79	-0.04
mcf	16	0.01	-0.23
gobmk	665	0.80	0.74
hmmer	101	-0.03	-0.24
sjeng	225	1.43	1.86
libquantum	55	0.24	0.02
h264ref	539	1.84	1.08
omnetpp	1669	0.07	0.02
astar	404	0.02	0.05
xalancbmk	5572	0.06	-0.03

Figure 6: Percentage improvement from GVL

Program	DSS	DCS	Base			GVL				Benefit	
			DCL	%MGD	%TLG	DCS	DCL	%MGD	%TLG	%RM	%RL
164.gzip	324K	115354	817295	99.5	90.1	115564	817765	99.5	90.5	-0.2	-0.1
175.vpr	18K	144798	1674822	0.2	0.2	144819	1680444	2.4	1.2	-0.0	-0.3
176.gcc	104K	24307	318739	8.2	4.8	24076	319507	8.2	4.8	1.0	-0.2
181.mcf	0.6K	50335	691167	0.0	0.0	50834	702752	0.3	0.2	-1.0	-1.7
186.crafty	928K	63261	424634	86.2	86.1	49123	334543	83.0	83.1	22.3	21.2
197.parser	50K	88758	841457	3.6	2.3	86069	823493	3.0	1.9	3.0	2.1
252.eon	130K	144644	998566	3.5	3.1	144554	994680	3.5	3.1	0.1	0.4
253.perlbnk	91K	222	1273	0.0	0.0	204	1348	0.0	0.0	8.1	-5.9
254.gap	117K	16251	391308	7.3	2.4	14613	375477	3.2	0.8	10.1	4.0
255.vortex	50K	44997	564978	3.9	2.5	37867	548690	1.4	0.8	15.8	2.9
256.bzip2	75K	36330	395646	1.9	1.8	35667	393915	1.1	1.3	1.8	0.4
300.twolf	37K	129810	1704611	4.3	3.0	132878	1763832	3.5	2.4	-2.4	-3.5

Key:	DSS	data segment size (kilobytes)
	DCS	total data cache misses (samples)
	DCL	total data cache miss latency (cycles)
	MGD	percentage of sampled misses to global data
	TLG	percentage of total latency to global data
	RM	percentage reduction in sampled misses due to GVL
	RL	percentage reduction in total latency cycles due to GVL

Figure 5: D-cache miss details

Column 3 in the Figure 6 table gives the performance benefits due to dynamic profile driven GVL over this baseline. We find that dynamic profile driven global variable layout gives improvement percentages ranging from 1% to 6%. The baseline for synthetic-profile GVL experiments is our default SPEC “base” configuration with all high level interprocedural optimizations enabled without dynamic profile data. In SPEC2000 benchmark suite, we find 4 benchmarks 176.gcc, 253.perlbnk, 255.vortex and 186.crafty showing performance improvements, with 186.crafty deriving the most benefit from this optimization. In the public-domain benchmark suite, the benefits seen from global variable layout are much smaller, with only 3 benchmarks (sjeng, gobmk and h264ref) showing performance improvement. In most of the benchmarks we studied, the dcache misses come from dynamically allocated data which are more amenable to optimizations such as structure layout optimizations[11] rather than global variable layout optimizations. Hence we find that GVL exhibits only a modest effect for SPEC integer benchmark programs. Column 4 gives the performance benefits due to synthetic-profile GVL over this baseline. We find that performance improvements are less than what we obtain with dynamic-profile GVL.

5.1 Data cache miss breakdown

Figure 5 shows more detailed information on data cache miss behavior for the SPEC2000 applications, obtained using the HP Caliper performance monitoring tool [12]. Caliper exploits the performance monitoring unit (PMU) hardware built into the Itanium chip, periodically sampling the hardware to collect data on the running target application. For this study, Caliper was configured to collect detailed information on data cache misses, including the specific instruction that caused the miss, the actual data address that caused the miss, and finally the miss latency in cycles. In this configuration Caliper targets only L1 misses, as the PMU only returns precise data addresses for these events.

However, one can derive that a miss comes from a higher level of the memory hierarchy from the latency values.

The data generated by Caliper are then post-processed to separate out cache misses to globals vs all other cache misses (to heap, stack, etc); the miss-to address from the Caliper data is compared with the virtual address range of the data segment from the application ELF file and bucketed appropriately.

For the benchmark 186.crafty, the data shows that a large fraction of cache misses before GVL are to global data, and that GVL gives a significant (22%) decrease in misses and in d-cache miss latency cycles overall.

5.2 Short data threshold

Our compiler also has an interprocedural “short data threshold” optimization, which operates during whole-program analysis. This optimization computes the size of the program’s statically declared data, along with an estimate of the linkage table overhead for the application, then arrives at a threshold N such that all data items of size N and below can safely be placed into the short (`.sbss`, `.sdata`) data sections without exceeding the size of the 22-bit GP offset used to address these items. Disabling the short data threshold optimization forces the compiler to consider all data whose size is greater than 8 bytes as non-short data.

Figure 7 shows the effect of turning off the short data optimization on the benefits obtained from GVL. We find that performance improvements from GVL are reduced when short data optimization is disabled. Using a lower short data threshold increases memory footprint due to the need for additional linkage table accesses, and it reduces the number of items that can participate in the cache-line centric “short” layout phase. This effect is very marked in the program 186.crafty, which has many “hot” data items that are greater than 8 bytes but less than 100 bytes in size.

Benchmark	% improvement from GVL	
	synthetic	dynamic
SPECint2000		
164.gzip	-0.07	0.29
175.vpr	-0.14	-0.08
176.gcc	1.27	0.00
181.mcf	0.34	0.00
186.crafty	3.86	2.63
197.parser	0.41	0.06
252.eon	0.31	0.00
253.perlbnk	1.41	0.66
254.gap	0.52	0.45
255.vortex	1.52	1.84
256.bzip2	0.28	0.38
300.twolf	0.05	-0.03
Public-domain applications		
perlbench	0.82	0.08
bzip2	0.03	0.02
gcc	0.64	-0.01
mcf	-0.08	-0.04
gobmk	1.38	0.24
hmmcr	0.11	0.61
sjeng	1.72	1.99
libquantum	0.17	0.03
h264ref	0.95	1.05
omnetpp	0.33	0.02
astar	-0.09	-0.02
xalancbmk	0.00	-0.08

Figure 7: Effects of disabling short data optimization

5.3 Short vs. long data

Figure 8 shows a comparison for the SPECint2000 benchmarks, comparing normal GVL with a run in which only short (less than a cache line in size) data items are subject to layout (long data items are not reordered). The data show that for most programs, the bulk of the beneficial effects of data layout come from reordering short items as opposed to long items.

Benchmark	% speedup
164.gzip	0.07
175.vpr	0.08
176.gcc	0.00
181.mcf	0.82
186.crafty	1.45
197.parser	0.15
252.eon	0.16
253.perlbnk	0.12
254.gap	0.21
255.vortex	0.00
256.bzip2	0.00
300.twolf	-0.08

Figure 8: Short+long GVL speedup vs. short-only GVL, SPECint2000

6. EXPERIMENTAL EVALUATION: OLTP

We evaluated the effects of our profile-driven variable layout technique by applying it to a commercial database application, the Oracle data server. This application is very large; the optimized Oracle binary has around 70000 functions, with a text segment size of approximately 110 megabytes for HP-UX Itanium. At runtime, the Oracle is organized as a collection of independent processes that communicate through very large shared memory segments; the execution profile of the server is heavily dominated by data cache

misses, primarily due to the accesses to shared memory. The size of the statically allocated globals for the application is comparatively small (400kbytes), however there are frequent accesses to the global data area.

Key compiler options for our particular Oracle executable included `+O2` (enable level 2 optimization), `+DD64` (enable 64-bit LP64 data model) and `+Oprofile=use` (enable feedback directed optimization including data layout). The execution profile we used was derived by preloading the database, then running 100k database transactions, which typically takes around 5 minutes.

We measured the performance of the Oracle application with and without data layout using a scaled-down copy of the TPC-C benchmark [26]. TPC-C is an OLTP benchmark meant to model a wholesale supplier managing orders, and is intended as a measure of database performance in a real-world commercial environment. A full-size TPC-C run can take as long as a day to run; the scaled down run in our testbed takes a little over 30 minutes. The testbed is comprised of two Integrity servers, a 4-processor HP `rx5640` machine running the database clients and a HP `rx4640` (configured with a single processor) running the server processes. Both machines have 16 gigabytes of main memory. The database itself has 10 warehouses; we use a total of 5 clients on the client side.

Enabling global variable layout at link time for the Oracle executable yielded an increased throughput in our testbed of approximately 2.5%. Although we have not repeated the same experiment with a full-scale TPC-C run, we have been told that the full-scale run reaps at least as much benefit (or more) from GVL than the scaled-down run.

7. RELATED WORK

Many previous studies have been made in the areas of data layout and instruction reordering for improving locality and to reduce cache miss and page fault penalties.

For scientific programs with regular loop patterns, traditional loop nest optimizations have proven to be effective [16, 1] and these transformations have been developed steadily over the last decades.

For irregular programs with complex data structures and references, compilers are trying to modify individual data structures and, for example, reorder structure fields, change a field's type, or break individual structures into pieces [15, 3, 4, 27, 24, 30, 22, 11]. These approaches target various different environments, from embedded applications up to server systems, and suggest multiple possibilities to transform data. Compilers use advanced scheduling algorithms to hide cache latencies and additionally insert pre-fetch instructions into the code stream to further improve the memory system performance.

Similar techniques are employed by dynamic compilers and runtime systems, which additionally employ the garbage collector to improve locality by collocating heap objects [23, 7, 10, 5]. A related technique has been proposed to collocate objects in pools on the heap for C/C++ programs [17].

Data transformations for eliminating false sharing misses in explicitly parallel programs have been discussed in several previous works, including Torrellas et al [25] and Jeremiasen et al [14]. The former work uses techniques based on trace analysis; the latter uses static analysis for explicitly parallel programs in which the synchronization and parallelization constructs are explicitly exposed to the compiler.

We believe that our techniques will be more suitable for use in global variable layout for very large applications, however, such as in a multiprocessor operating system kernel.

There are two papers presenting work close to ours, Calder et al [2] and Haber et al [8]. Calder suggests a unified approach and targets globals, stack and heap variables, as well as constants, which correspond to the read-only data presented earlier. Similar to our approach, they build an affinity graph to guide layout decisions. However, their heuristics and constraints in terms of data grouping and alignment differ substantially. They rely on a special profiler and linker to make the system work and do not employ an alternative full IPA for cases where no profile is available.

Haber et al also describe a data layout transformation. They perform this transformation in a post-link tool and gain performance benefits from eliminating GP relative references. In our infrastructure, the short-data runtime architecture and compiler optimization seem to subsume most of these benefits, which on Itanium generally appear to be lower than described in their work. We could conceivably perform additional grouping of data objects to try to further reduce the usage of GP relative addressing operations, however, as we have shown with the analysis of the short data optimization, we don't expect any further tangible performance benefits.

8. FUTURE WORK

We plan to extend this work to target parallel applications running on shared-memory multiprocessors with hardware cache coherence, with the goal of reducing false sharing misses. If a heavily-written global scalar "W" is positioned by GVL so that it is in the same cache line as an infrequently written but heavily read scalar "R", the writes to "W" can invalidate useful cached copies of "R" in other processors even though those processors will never consume the new value of "W".

Our GVL analysis phase can address this problem by virtue of the fact that we separately track reads and writes for each data item. For parallel applications, during the layout phase of GVL, if we encounter a data item that is heavily written (write count exceeds a heuristically determined threshold), we place this data item into a separate cache line; we call this process "segregation". We can insure that a heavily written scalar is placed into a separate cache line by over-aligning it (giving it an alignment requirement equal to the size of the line for the cache used to enforce coherence) and padding it. This is similar to the transformations discussed in previous studies [25, 14].

Our target application for this work is the HP-UX operating system kernel running the SDET benchmark on large (64-way and 128-way) Integrity multiprocessor servers. In performance analysis studies, kernel developers have identified cases where false sharing is causing additional cache misses due to the unoptimized data layout of existing kernels.

9. CONCLUSIONS

In this paper we have focused on a global variable re-ordering technique, GVL, that helps reduce data cache miss penalties for programs running on high-performance processors with cache memories. We have presented and evaluated two complementary compiler techniques for GVL, the first

driven primarily based on profile feedback, the second using interprocedural analysis and optimization techniques. We have demonstrated the effectiveness of our techniques through experimental studies, with both the SPEC integer benchmarks and with a large commercial database application.

10. ACKNOWLEDGEMENTS

The authors would like to thank the entire HP-UX Integrity compiler, linker and tools team, for their efforts in creating an industrial-strength optimizing compiler for Itanium (without which this paper would not have been possible). Thanks to Chris Ruemmler and Dan Truong, for help with GVL work on the HP-UX kernel. Thanks to Dmitry Mikulin for contributions to the linker portion of the profile-driven GVL implementation, and thanks to Luis Lozano and Kerch Holt for help gathering the Oracle timings. Finally, special thanks to Tor Ekqvist and Bill Habeck for creating the Oracle client/server testbed used for our internal testing and performance tuning efforts.

11. REFERENCES

- [1] D. F. Bacon, J.-H. Chow, R. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. In *CASCON '94: Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research*, page 3. IBM Press, 1994.
- [2] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.
- [3] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 13–24, New York, NY, USA, 1999. ACM Press.
- [4] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 1999. ACM Press.
- [5] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ISMM '98: Proceedings of the 1st International Symposium on Memory Management*, pages 37–48, New York, NY, USA, 1998. ACM Press.
- [6] T. Ekqvist. Squeezing performance out of the Intel Itanium architecture. HP World 2003, http://h71028.www7.hp.com/enterprise/downloads/Squeezing_performance_Itanium_100803.pdf.
- [7] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 177–186, New York, NY, USA, 1993. ACM Press.

- [8] G. Haber, M. Klausner, V. Eisenberg, B. Mendelson, and M. Gurevich. Optimization opportunities created by global data reordering. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 228–240. IEEE Computer Society, 2003.
- [9] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.
- [10] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 69–80, New York, NY, USA, 2004. ACM Press.
- [11] R. Hundt, S. Mannarswamy, and D. R. Chakrabarti. Practical structure layout optimization and advice. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2006. IEEE Computer Society.
- [12] R. Hundt. HP Caliper: A framework for performance analysis tools. In *Concurrency, IEEE*, pages 64–71, Washington, DC, USA, 2000. IEEE Computer Society.
- [13] Intel. Itanium Software Conventions and Runtime Architecture Guide. <http://www.intel.com/>.
- [14] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, 1995.
- [15] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. In *ACM Transactions on Programming Languages and Systems, v.22 n.3*, pages 490–505, 2000.
- [16] M. S. Lam and M. E. Wolf. A data locality optimizing algorithm. *SIGPLAN Notices*, 39(4):442–459, 2004.
- [17] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, New York, NY, USA, 2005. ACM Press.
- [18] S. McFarling. Program optimization for instruction caches. In *ASPLOS-III Proceedings - Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.
- [19] S. Moon, X. D. Li, R. Hundt, D. R. Chakrabarti, L. A. Lozano, U. Srinivasan, and S.-M. Liu. SYZGYG - a framework for scalable cross-module IPO. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 65, Washington, DC, USA, 2004. IEEE Computer Society.
- [20] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27:9, pages 62–73, New York, NY, 1992. ACM Press.
- [21] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):16–27, June 1990.
- [22] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *Transactions on Embedded Computing Sys.*, 2(2):186–218, 2003.
- [23] M. B. Reinhold. Cache performance of garbage-collected programs. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 206–217, New York, NY, USA, 1994. ACM Press.
- [24] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 140–153, New York, NY, USA, 2002. ACM Press.
- [25] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.
- [26] Transaction Processing Performance Council. TPC Benchmark C Standard Specification, 1990.
- [27] D. N. Truong, F. Bodin, and A. Sez nec. Improving cache behavior of dynamically allocated data structures. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 322, Washington, DC, USA, 1998. IEEE Computer Society.
- [28] UNIX System Laboratories. *System V Application Binary Interface*. UNIX Press, 1992.
- [29] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 1–11, November 1994.
- [30] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI '04: Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.