Profile-based Speculation

1. Introduction
Modern processors often employ speculation to improve instruction level and thread level parallelism. They predict the outcome of data and control decisions, and speculatively execute the operations and commit results only if the original predictions were correct. Mis-speculation usually causes expensive recovery overhead, and therefore processors will need some form of guidance to balance the benefits of speculation against recovery overhead.

Profiling is an effective technique to guide highly successful speculations. A profile is the aggregate information about the behavior of a program collected from past execution. Depending on the hardware speculation mechanism, various profiled-based speculations can be used.

Control flow profile is often used to order basic blocks or form regions to improve execution path speculation [Young97, Hank95]. Control flow profile is also crucial to decide load speculation passing conditional branches to improve instruction level parallelism [Mahlke93]. Cache profile can be used to identify load instructions that frequently miss cache so they can be speculatively prefetched into cache [Mowry00][Lebeck94][Adl-94], or prevented from occupying cache storage if they cannot be cached profitably [Wu03]. Value profile has been successfully used to guide stride prefetching [Wu02], speculative reuse [Wu01], and speculative multi-threading [Du04].

In this Chapter, we describe a few common profiles that are used in practice, techniques for profile collection, examples of profile-based speculations, and the issues with successful profile-based speculations.

2. Commonly-used profiles
We may loosely classify profiles into the following categories:

- Control flow profile: block profile, edge profile, and path profile, and call graph profile.

- Memory profile: Affinity profile, cache profile, hot data stream profile, alias profile, and stride profile

- Value profile: top-value profile and reuse profile

2.1. Control flow profile
Historically, profile-based techniques mostly explore the control flow profiles. A control flow profile provides aggregate information about the properties of the program’s structure, such as how frequently a block is executed, how likely a control flow edge takes, how often a control flow path is executed, and how frequently a function calls another function.
Block profile counts the number of times each basic block is executed, and thus provides relative importance about various areas of a program to guide compiler optimizations. Edge profile captures the frequencies of control transfer between basic blocks, which are often used by compiler to connect basic blocks to enlarge optimization scope. Block profile can be used to derive edge profile although in certain cases that cannot be done accurately. Edge profile can always be used to calculate the block profile [Ball92].

For blocks with multiple predecessors and successors, edge profile sometime is not sufficient to form highly probable execution paths. Path profile directly determines the frequency that each program path is executed. A program path is usually acyclic, such as a path from a loop entry to a corresponding loop exit or from a function entry to a function return without going through a backedge of any loop [Ball96]. Cyclic paths [Young98] and inter-procedural paths [Glenn97] can also be profiled and useful.

A special inter-procedural control flow profile is the call graph profile, which counts the number of times one procedure calls the other. Call graph profile is especially useful for procedural placement to improve instruction spatial locality [Annavaram03, Kalamatianos99, Wu92, Hall92]. Notice that the frequency that a function calls another function is not simply the block frequency of the basic block containing the function call. Complication arises when conditional call and indirect call are encountered.

2.2. Memory profile
As memory stall becomes more and more dominant in program performance, many forms of memory profiles begin to emerge. Memory profile characterizes memory reference patterns from past execution, which can be used to speculatively perform memory operations and hide memory stalls. Memory profile has many different forms, such as cache profile, affinity profile, alias profile, and stride profile.

Cache profile [Lu-03, Mowry00, Lebeck94, Adl-tabatabi04] identifies memory references (e.g. loads) that cause the most cache misses. Knowing which loads generating most the cache misses, the speculative execution may fetch the loaded memory in advance to overlap the cache misses with other executions.

Alias profiling [Lin03] calculates probabilities that memory operations may conflict with each other to support speculative optimizations that rely on accurate alias information.

Affinity profile [CHILIMBI-99] maps each pair of field references to a numeric number indicating how often they are both referenced within a pre-defined interval. A field reordering heuristic can then use this pair-wise relationship to place fields according to their temporal affinity.

Temporal data reference profile [Chilimbi02] identifies hot data streams, which are data reference sequences that frequently repeat in the same order, to support runtime detection and speculatively prefetch of hot data streams.

Stride profile recognizes regular reference patterns of load instructions to guide prefetching the loads inside loops [Wu02]
2.3. Value profile

A computer operation, such as a load instruction, a subroutine, etc, ultimately produces some outcome values. If the values can be predicted before the operation is executed, the program can run much faster by using the outcome value without waiting for the operation to complete. Value profile [Calder-97, Gabbay97] provides an effective method to collect the mostly likely outcome values.

Value profile may be collected for individual instructions. For example, a multiplication or divide instruction can be profiled to identify the mostly like operand values. If one of the operand is a runtime constant, the long latency operation can be speculatively replaced by a few of simple instructions [Wu91].

Value profile may also be collected at region level. A region of code can be profiled to identify the most likely live-in and live-out values. If the region, such as a loop, has a few dominant live-in and live-out values, it is then a good candidate for speculative computation reuse [Wu01].

3. Profile collection

The simplest method to collect profile is static program analysis [Ball93, Wagner94, Wu94]. Profile accuracy can be improved significantly with instrumentation and a separate run of the program with a training input [Chang91]. However, some profile, such as cache profile, is very expensive to collect by instrumentations. Sampling-based hardware performance monitor becomes popular to collect profiles related to the micro-architectural events, such as cache miss and branch miss prediction. Special hardware supports such as the profile buffer and the software-hardware collaborative approach, can obtain control flow profiles with very little overhead to support runtime optimizations ([Conte96], [Daisy], [Wu99], [Subr-01]). Some compilation system uses multiple profiles to guide its optimizations. For example, the StarJIT Java system uses instrumentation to collect control flow profile for classical optimizations, and uses performance monitoring to collect cache profile for memory optimizations [Adl-03].

3.1. Static analysis

By analyzing program syntax and semantics, it is often possible to determine how often a branch may be taken. This eliminates the drawbacks of other profiling techniques, such as programmer’s intervention in for a separate profiling phase, or the overhead in the dynamic profiling and optimizations. [Ball93] uses a set of heuristics to prediction branch directions, such as “loop branch heuristic” predicts that the loop back edge will be taken, “pointer heuristic” predicts that a comparison of the pointer against null will fail, and “opcode heuristic” predicts that a comparison of an integer for less than zero will fail, etc. [Wagner-94] uses a value of 0.8 as the branch probability for a branch predicted to take, and 0.2 as the branch probability for a branch predicted not to take. [Wu-94] uses the Dempster-Shafer theory to systematically convert the predictions from all the heuristics to obtain the branch probability information useful to the compiler. For example, the branch probabilities can be used to obtain a relative ordering of the basic blocks. Among the 20% of the hottest blocks, more than 80% of them are identified correctly by static program analysis [Wu-
Almost all commercial compilers have static analysis module to provide profile information for optimizations.

3.2. Instrumentation

Instrumenting a program to collect profile data involves the insertion of code to increment a set of counters that corresponds to a subset of the interested program entities. For example, to collect Control Flow Graph (CFG) edge profile, a compiler first counts the number of edges in the CFG for a function and assigns a unique identification number to each edge. This identification number is then used as an index into an array of execution counters. For architecture with predicated execution, the instrumentation code can be predicated with the appropriate condition indicating whether or not the branch was taken. In this manner, a compiler is able to avoid inserting additional basic blocks into the CFG, while inserting instructions “on the edge.” Furthermore, the inserted instructions can then be scheduled in parallel with other instructions in the basic block, thereby reducing the cost of executing the instrumentation code.

Instrumentation may incur significant runtime overhead for collecting the profiles. Compilers often try hard to reduce the amount of instrumented code. For example, [Knuth73] suggests a technique to locate the minimal set of blocks to place instrumentation. From the frequencies of the minimal set of blocks, the block frequency for all the blocks can be derived. Similarly, to collect the edge profile for a function, only the edges on the cords of a minimal spanning tree of the control flow graph need to be instrumented. The edge frequency information for other edges can be derived from the instrumented edges [Ball94]. [Ball96] also devised a clever technique to instrument a subset of the control flow graph edges with simple operations to efficiently collect the frequencies for all the executed paths. Still programs with instrumentation for block/edge/path profiling typically run about 10% to 40% slower than without the instrumentation. The increased execution time may affect the behavior of real time and reactive systems.

Instrumentation can also be used to collect other more detailed profiles (e.g. cache, aliasing, value, etc). For example, to collect cache profile, code can be inserted before load and store instructions to simulate the effect of the memory reference on cache and record the cache miss statistics. The program with the instrumentation usually runs many times slower than without the instrumentation. The profiling overhead can be reduced via either sampling based instrumentation (e.g. the Burst Tracing [Hirzel-01]) or the hardware performance monitoring unit (PMU).

3.3. Hardware performance monitoring

One way to reduce profiling overhead is to use infrequent sampling of program PCs to identify frequently executed code. [Zhang-97] uses a time interrupt- based approach to collect sampled frequency of executed code blocks. By accumulating the number of samples occurred at each code block, the relative order of the code blocks can be determined. The relative ordering is used to layout basic blocks in an offline process. They reported an overhead of about 0.3% due to the interrupt and the saving of the sampled data.

More sophisticated performance monitoring uses dedicated performance monitoring unit (PMU) to record
detailed microarchitectural events for supporting instruction-level profiling. For example, the Itanium hardware [Intel02] attributes events like branches, cache and TLB misses to individual instructions so that software can know exactly where to optimize in the program. IPF implements three hardware structures for this purpose: Branch Trace Buffer (BTB), Instruction Event Address Register (I-EAR), and Data Event Address Register (D-EAR). By performing statistical sampling on these structures, instruction-level profiling can be done at a low cost.

The BTB captures the information about the last 4 to 8 branches, including the branch’s PC, branch target’s PC, and mispredict status. By taking enough samples of the BTB, control flow graph for those frequently execution code can be constructed, and hot traces can be identified for optimizations. Also the branches that are frequently mispredicted can be identified for branch optimization (such as predication).

The I-EAR records that information about the last I-cache and I-TLB miss, including the instruction PC and miss latency in cycles, and which level serviced the I-TLB miss (L2 I-TLB, VHPT, or software). D-EAR is similar, it records the miss instruction PC, data address, miss latency in cycles, which level serviced the D-TLB miss (L2 D-TLB, VHPT, or software). Once the instructions that cause the most cache misses are identified, instruction or data prefetching can be used to improve program performance.

3.4. Special hardware

Although PMU can be viewed as the general hardware for collecting instruction-level profiling information, they are primarily designed for performance analysis and tuning. Here we discuss special hardware targeting special optimization needs. For example, The DAISY VLIW processor [İbcioglu01] contains hardware support for profiling in the form of an 8K entry 8-way set associative (hardware) array of cached counters indexed by the exit point id of a tree region. These counters are automatically incremented upon exit from a tree region and can be inspected to see which exit points are consuming the most time. If it is found that the time spent in a tree region from the root to an exit point is greater than a threshold of the total cycles spent in the program, the tree region is extended at the exit point. Tree region extension allows translations using a significantly higher ILP and larger window size. Thus, if there are parts of the code which are executed more frequently than others (implying high reuse on these parts), they will be optimized very aggressively.

As another example, [Chen03] uses a hardware profiler to determine dynamic dependency behavior (load dependency analysis) and buffering requirement (speculative state overflow analysis) to identify the best speculative thread loops (STL) to parallelize dynamically. The load dependency analysis looks for inter-thread dependencies for a STL. A store timestamp is recorded on a memory or local variable store, and retrieved on a subsequent load to the same address. The store timestamp is checked against thread start timestamps to determine if an inter-thread dependency arc exists to the previous thread (t-1), or an earlier thread (< t-1). If an inter-thread dependency arc is detected, the arc length, the difference between the current time and the store timestamp, is recorded. While many dependency arcs may exist between any two given speculative threads, the hardware only record the critical arc, which is the shortest arc and limits...
parallelism between the threads. The speculative state overflow analysis checks that speculative state for a
STL can fit within the L1 caches and store buffers. Each L1 data cache line includes additional tag bits to
record a processor’s speculative read state. Each speculative store buffer, with L1 cache line sized entries,
collects all speculative heap writes made by a processor. Dropping a L1 cache line with speculative state or
overflowing a store buffer forces a speculatively executing thread to stall until reads or writes can be
performed safely.

The hardware profiler leverage the store buffer available in the Hydra chip multiprocessor for profiling
purpose and adds less than 1% to the total transistor count of Hydra. Benchmarks experience an average
7.8% slowdown during profiling. Without hardware support, the execution slows over 100x when applying
the analysis with software alone.

3.5. Software-hardware collaborative profiling

The approach in [Wu-01] achieves low profiling overhead and high profile accurate by combining two
collaborative techniques: powerful compiler analysis that inserts minimal profiling instructions; and
hardware that asynchronously executes profiling operations in free slots in user programs running on wide-
issue processors like IA-64.

With the collaborative method, the compiler first analyzes the user program to determine the minimal set of
blocks that need profiling (e.g., using the Knuth algorithm [Knuth93]). Next, it allocates profile counter
space for each such block and inserts an “iniprof” instruction in the function entry block to let the hardware
know the starting address of the profile counter space. Then it inserts a “profile ID” instruction in each
profiled block to pass profiling requests to the hardware. The hardware derives the profile counter address
from the information passed by the “iniprof” and “profile ID” instructions and generates profile update
operations. These update operations are then executed when free slots become available during the user
program execution.

Since most of the profiled blocks contains a branch instruction, the “profile ID” instruction can be avoided
by encoding the ID into the branch instruction. By using an 8-bit ID field in branch instructions, most
“profile ID” instructions can be eliminated. For large functions that require more profile ID’s than the 8-bit
ID field can hold, the compiler employs a graph partitioning algorithm to partition the CFG into single-
entry regions, such that the ID values in each region can be encoded into the 8-bit ID field. It uses an
“setoffset” instruction in each region entry block to pass an offset value of the region to the hardware.

The new profiling technique enables programs with profiling run almost as fast as without profiling. For
example, a less than 0.6% overhead for SPECint95 benchmarks is observed while collecting complete
block profile without sampling.

4. Profile usage models

Profile-based optimization typically operates in two phases, a profile phase and an optimization phase that
uses the profile. Depending on when the profile is collected, we often see two different usage models: the compile-time profiling model, and the runtime profiling model. There are also recent proposals that suggest continuous profiling for phase detection and re-optimization in dynamic systems.

4.1. Compile-time profiling

The profiling phase in static compiles is a separate compilation pass. During the profiling pass, the program is compiled and run to collect profile information. After the profile is collected, the program is compiled the second time to use the profile. This approach is straightforward to implement and almost all optimizing compilers support it. However, this two-pass process is inconvenient to use, as it not only complicates the software development but also the process for validation and performance evaluation.

Another issue with the profiling pass is the training input required for the profiling run. It is a challenge to obtain representative training input to collect profile. If the training input is not representative of the actual input of the program, the program may be optimized with incorrect profile. For example, many commercial programs have multiple functions to suit various field requirements. It may be more effective to specialize the program with real-world workload at customer sites rather than collecting profile by the program developer.

4.2. Two-phase runtime profiling

Dynamic systems run real-world workload at customer sites, and are normally not allowed to have separate trial runs to collect profiles. Consequently, most dynamic optimization systems (e.g. IA32EL [Baraz03], Transmeta [Dehnert03], Daisy [Ebcioğlu97] and Dynamo [Bala99]) use a two-phase approach to identify and optimize frequently executed code. In the first phase (profiling phase), blocks of code are interpreted or translated without optimization to collect execution frequency information for the blocks. In the second phase (optimization phase), frequently executed blocks are grouped into regions and advanced optimizations are applied on them.

The time to collect the profile is counted as the execution time of the program, and any performance gain from profile-based optimizations has to amortize the profiling overhead first. Consequently the system needs to manage the profiling phase sophisticatedly so that just enough profile are collected and majority of the execution is with the optimized code. Furthermore, supporting both the profiling and optimization phases while the real program is running involves significant bookkeeping effort. For example, the interpreter and the optimizer need to work together so the transition between interpreted execution and optimized execution occurs smoothly.

The benefit of the runtime profiling approach is that the profiling phase uses the early part of the same input to collect profile for late execution. This requires significantly less profiling effort to obtain similar or more accurate profiles than the training input. For example, [WU-94] shows that the two phase runtime profiling only needs to run each basic block for a few thousands of times and obtain similar or more accurate branch probability profiles than that obtained from the training input. To profiling each block for
a few thousands of times represents about 0.5% of the profiling operations required for the training run (for the SPEC2000 benchmarks).

4.2. Continuous profiling
Recent studies [Barnes02, Kistler03, Sherwood03] have shown that some programs exhibit phases behavior. For those programs, a single profiling phase is clearly unable to respond to the phase changes. Continuous profiling collects profile even after the program is optimized. When the executing program changes characteristics significantly from the previous profile, the program is re-optimized with the new profile.

Since the optimized program needs to be continuously profiled, the profile must be collected without slowing down the optimized execution. [WU-94a] inserts a few instructions in each loop to continuously collect loop trip count information for advanced loop optimizations. The instrumented code incurs about 0.5% of performance penalty. The advanced loop optimization needs to improve the performance for more than 0.5% for the profiling effort to pay off.

PMU-based profiling may also be used for continuous profiling. Although the periodic sampling and profile processing incur overhead, PMU-based profiling can avoid inserting code into optimized programs so the code quality won't suffer. Furthermore, the continuous profiling aims at detecting significant phase changes, and therefore can tolerate certain profile inaccuracy. As the result, the sampling frequency can be relatively small to reduce profiling overhead.

[Lu-03] presents a dynamic optimization system with PMU to continuously detect instructions with high data cache misses. The overhead of this prefetching scheme and the dynamic optimization is very low due to the use of sampling on the Itanium processor's performance monitoring unit. Using this scheme, they can improve runtime performance by as much as 57% on some SPEC2000 benchmarks compiled at O2 for Itanium-2 processors. For binaries compiled at O3 with static prefetching, the system can improve performance by as much as 20% for some SPEC2000 benchmarks.

However, for optimizations other than data prefetching, it is still open whether the continuous profiling and optimization for capturing phase changes is able to improve performance significant enough to offset the overhead of continuous profiling and re-optimization [Kistler03].

5. Profile-based ILP speculations
To improve instruction level parallelism (ILP), it is important to have a large span of program visible to the instruction scheduler. The processor uses branch prediction hardware to speculatively schedule future instructions for parallel execution. The compiler uses profile information to speculatively move instructions across branches to increase ILP. Examples of such speculative transformations are the trace scheduling [Fisher81], hot-code optimizations, and the superblock optimizations. The compiler may also use profile to arrange code and data [Ramirez01] so that the control and data flow can be easily speculated
4.3. Trace scheduling

Trace scheduling was first implemented in the Multiflow compiler for a VLIW processor to improve instruction level parallelism for control-intensive programs. The compiler uses edge profile information to select the most like path, or “trace” that the code will follow during execution. The trace is then treated as if it were free of conditional branches during instruction scheduling. The aggressive instruction scheduling may cause code motions which could cause logical inconsistencies when branches off-trace are taken. The compiler inserts special compensation code into the program graph on the off trace branch edges to undo those inconsistencies. [Freudenberger94] shows that trace scheduling improves the SPECmark rating of SPEC89 suite by 30% over basic block scheduling. Restricting trace scheduling so that no compensation code is required improves the rating by 25%.

The possibility that a trace may have side entries to the middle of trace complicates the scheduling task significantly. A superblock is essentially a trace without side entries. Any trace with multiple entries can be converted to a superblock by an optimization called tail duplication. Profile-guided superblock optimization and scheduling [Hwu-99] is now widely used due to its simplicity and performance improvement.

4.4. Hot-cold optimizations

In a superblock, a loaded value before a branch may be only used after a side-exit is taken. Since the side exit is unlikely to be taken, the load instruction is often unnecessarily executed. In general, a dynamic instruction trace often contains many unnecessary instructions that are required only by the unexecuted portion of the program. The spirit of hot-code optimization is to use profile information to partition program into frequently executed (hot) and infrequently executed (cold) parts. Unnecessary operations in the hot portion are removed, and compensation code is added on transition from hot to cold as needed. About 3-8% reduction in path length beyond conventional optimization is reported for large Windows NT applications [Cohn-96].

With hardware support, this idea can be push further. FastForward [Li-00] is a collaborative compiler and hardware technology that aggressively optimizes predicted hot paths and employs hardware and compiler transformation to handle the violation of prediction.

A FastForward region starts with a superblock (or hyperblock) with the execution reaching from the entry to the main exit with a high probability (based on edge or path profile information). The compiler then forms a FastForward Region (FFR) by removing the low-probability side-exit branches. For each removed branch, the compiler inserts an assert instruction [Patel99] inside the FFR or an abort in the original code.

On Itanium processor, each assert instruction takes the predicate of a removed branch as the sole operand while the original branch is omitted. It fires when the predicate (i.e., infrequent condition) becomes true. A
fired assert instruction will terminate the execution of the containing FFR region, and when that occurs, the partial result computed by the FFR is discarded and the original code is executed.

After the infrequently executed code is removed, the FFR regions often have significantly simplified data and control flow graphs since there are no side entries and side exits. A compiler can optimize such regions much more effectively. FastForward regions enable the compiler to explore more optimization opportunities that were not present in the original program. For example, the following are a few situations:

- More dead code removal. Since many low-probability side exits are removed, a computation that is live out only at the side exits becomes dead. Furthermore, when the low-probability branch is removed, the code that defines the branch condition may become dead.

- Better scheduling. Simplified control flow removes the need for speculation and compensation code in many cases. In addition, large regions provide a large scope for scheduling and thus yield better IPC (instructions per cycle).

- Better register allocation. Simplified data and control flow graphs reduce the number of live variables and thus the register pressure.

- More predication. By removing infrequent side entries and side exits, more predication opportunities exist to explore the modern architecture features [Intel02].

The classification of code as hot or cold depends on its execution profile. An important consideration is determining a threshold for the probability that control will flow from a hot region to a cold region during program execution. This threshold should be as small as possible.

Experimental result shows about 10% reduction in path length beyond conventional optimization for SPEC95 integer benchmarks.

Notice that partial redundancy elimination (PRE) and partial dead-code elimination (PDE) [Gupta97] could achieve some of the benefit of FastForward. However, our experience indicates that PRE and PDE are ineffective and are very complex to implement. FastForward simplifies data and control flow to a single basic block so redundancy and dead code are easily removed. In addition, simplified data and control flow brings many benefits for other compiler optimizations, such as register allocation, instruction scheduling, and if-conversion.

4.5. Code layout

Code layout optimization [Pettis90] improves ILP by reducing pipeline stalls due to instruction cache misses. Typical basic block size is smaller than the icache line size. Arranging basic blocks such that the most likely execution path is along the fall-through target of the branches will help improving the icache locality. Also, many of the processors employs instruction prefetching such that when the current instruction is executed, the next a few cache lines of code is prefetched into the instruction cache.
Arranging a function close to the function that calls it will improve the chance that the called function will be already prefetched into icache when it is called. Furthermore, the average function size is significantly smaller than page size. Arranging a function close to the function that calls it will improve the chance that the called function will be in the same page as it caller. This will reduce the dynamic memory footprint and potentially reduce ITLB misses as well as bus traffic [Wu-92]. For large functions, function splitting can separate cold portion of the function from the hot portion to improve the spatial locality of frequently executed code. Most of the code layout algorithms use profile to determine block, branch and function call frequencies.

[Ramirez-01] provides a detailed study of profile-driven compiler optimizations to improve the code layout in commercial workloads with large instruction footprints. Their results show that code layout optimizations can provide a major improvement in the instruction cache behavior, providing a 55% to 65% reduction in the application misses for 64-128K caches. This improvement primarily arises from longer sequences of consecutively executed instructions and more reuse of cache lines before they are replaced. It is also shown that the majority of application instruction misses are caused by self-interference. However, code layout optimizations significantly reduce the amount of self-interference, thus elevating the relative importance of interference with operating system code. Furthermore, it shows that better code layout can also provide substantial improvements in the behavior of other memory system components, such as the instruction TLB and the unified second-level cache. The overall performance impact of the code layout optimizations is an improvement of 1.33 times in the execution time of workload.

4.6. Data layout

Traditionally loop transformation (e.g. tiling) and data layout have been able to reduce data cache miss and DTLB miss via program analysis and transformation for numeric programs [Park03]. For general-purpose programs, however, compiler analysis usually is insufficient to make data layout decisions. As the result, compiler resorts to profile to determine data layout optimization.

Data-layout optimizations synthesize a layout with good spatial locality generally by (i) attempting to place contemporaneously accessed memory locations in physical proximity (i.e., in the same cache block or main-memory page), while (ii) ensuring that frequently accessed memory cells do not evict each other from caches. Finding an optimal memory layout is intractable [PETRANK02]. Most techniques use profile information to guide the heuristic data layout optimizations. For example, field reordering optimization [Chilimbi-99] first collects field pair-wise affinity profile to map each pair of fields to a frequent value of the number of times the two fields are accessed in close time intervals. The profile information is then used to construct an affinity graph of fields, where nodes are fields and arcs are marked with frequency values. Based on graph clustering algorithms, the optimization clusters fields with high temporal affinity into the same cache block. This approach can also be used to reorder global variables by treating all the global variables as the fields of a single data structure [Calder98].

Similar to function splitting, large data structure may be split into hot and code sub-structures.
[Chilimbi99] uses field access frequency profile to identify hot fields. If the collection of hot fields is smaller than the cache line size, splitting the hot fields into a separate structure can help reduce cache misses.

[Rubin-02] proposes a profile analysis framework for data-layout optimizations. The framework finds a good layout by searching the space of possible layout, with the help of profile feedback. The search process iteratively prototypes candidate data layouts, evaluating them by simulating the program on a representative trace of memory accesses.

5. Profile-based data prefetching

Data prefetching is a speculation technique to overlap cache miss latency with useful computation. Hardware schemes requires significantly silicon budget. Compile-time data prefetch however is inefficient for general purpose programs. A recent trend is to use profile to guide effective data prefetching.

5.1. Stride prefetching

Irregular programs such as the SPECINT2000 benchmarks contain many irregular data references caused by pointer-chasing code. Irregular data references are difficult to prefetch, as the future address of a memory location is hard to anticipate by a compiler or hardware. However, many SPECINT2000 programs contain important references with stride patterns. For example, the SPECINT2000 197.parser benchmark has code segments as shown in Figure 1. The first load at S1 chases a linked list and the second load at S2 references the string pointed to by the current list element. The program maintains its own memory allocation. The linked elements and the strings are allocated in the order that is referenced. Consequently, the address strides for both loads remain the same 94% of the time.

```
for (; string_list != NULL; string_list = sn) {
    S1: sn = string_list->next;
    S2: use string_list->string;
    other operations;
}
```

Figure 1. Pointer-chasing code with stride patterns

SPECINT2000 benchmark 254.gap also contains pointer reference loads with stride patterns. An important loop in the benchmark performs garbage collection. A simplified version of the loop is shown in Figure 2. The variable s is a handle. The first load at the statement S1 accesses *s and it has four dominant strides, which remain the same for 29%, 28%, 21%, and 5% of the time, respectively. One of the dominant stride occurs because the increment at S4. The other three stride values depend on the values in (*s&~3)->size added to s at S3. The second load at the statement S2 accesses (*s & ~3)->ptr. This access has two dominant strides, which remain constant for 48% and 47% of the time, respectively. These strides are mostly affected by the values in (*s&~3)->size and by the allocation of the memory pointed to by *s.
while (s < bound) {
    if (*s & 3 == 0) { // 71% time is true
        access (*s & ~3)->ptr
    } else if (*s & 3 == 2) { // 29% time is true
        s = s + *s;
    } else { // never come here
        other operations;
    }
}

Figure 2. Irregular code with multiple stride patterns

Static compiler techniques, however, cannot easily discover the stride patterns in irregular programs. Pointer references make it hard for a compiler to understand the stride patterns of the load addresses. Also, the stride patterns in many cases are the results of memory allocation and compiler has limited ability to analyze memory allocation patterns. Without knowing that a load has stride patterns, it would be futile to insert stride prefetching, as doing so will penalize the references not exhibiting the prescribed strides.

[Wu02] develops an efficient profiling method to discover loads with stride patterns. It integrates the stride profiling into the traditional frequency-profiling pass. The combined profiling pass is only slightly slower than the frequency profiling alone, as only loads in loops with high trip counts (e.g. > 128) are profiled. The resulting stride profile is used to guide compiler prefetching during the profile feedback pass.

The example in Figure 3 illustrates our profiling and prefetching techniques. Figure 3 (a) shows a typical pointer-chasing loop. Assume that the data address of the load reference P->data at L is P. The compiler instruments the loop for both block frequency and stride profiling as shown in Figure 3 (b). The operations freq[b1]++ and freq[b2]++ collect block frequency information for the loop pre-head block b1 and the loop entry block b2. The conditional assignment "pr = (r2/r1) > 128" sets the predicate pr to true when the loop trip count is greater than 128, and false otherwise. The profiling runtime routine strideProf is guarded by the predicate pr and it is actually invoked only when the predicate pr is true. After the instrumented program is run, the stride profile is fed back and analyzed. The profile could indicate that the load at L has the same stride, e.g. 60 bytes, 80% of the time. In this case, the compiler can insert prefetching instructions as shown in Figure 3 (c), where the inserted instruction prefetches the load two strides ahead (120=2*60). The compiler decides the number of iterations ahead using information such as the size of the loop and the stride value. In case the profile indicates that the load has multiple dominant strides, e.g. 30 bytes 40% of the time and 120 bytes 50% of the time, the compiler may insert prefetching instructions as shown in Figure 3 (d) to compute the strides before prefetching. Furthermore, the profile may suggest that a load has a constant stride, e.g. 60, sometimes and no stride behavior in the rest of the execution, the compiler may insert a conditional prefetch as shown in Figure 3 (e). The conditional prefetch can be implemented on Itanium using predication.
The stride profile guided compiler prefetching obtains significant performance improvement for the SPECINT2000 programs running on real Itanium machines. For example, it achieves a 1.59x speedup for "181.mcf", 1.14x for "254.gap", 1.08x for "197.parser". These performance improvements, with an average of 7% for the entire benchmark suite, are significant for highly optimized SPECINT2000 programs running on real machines.

5.2. Hot data stream prefetching

[Chilimbi-02] developed a hot data stream prefetching scheme to prefetch for general programs. It operates in three phases. First, the profiling phase gathers a temporal data reference profile from a running program with low-overhead. Second, the profiling is turned off and a fast analysis algorithm extracts hot data streams, which are data reference sequences that frequently repeat in the same order, from the temporal profile. Then, the system dynamically injects code at appropriate program points to detect and prefetch these hot data streams. For example, given the hot data stream “a b a c d a e”, the optimizer inserts code to detect the data references “a b a”, which is called the prefix of the hot data stream, and prefetches from the addresses “c d a e”. Ideally, the data from these addresses will be cache resident by the time the data references takes place, avoiding cache misses and speeding up the program. Finally, the process enters the hibernation phase where no profiling or analysis is performed, and the program continues to execute with the added prefetch instructions. At the end of the hibernation phase, the program is deoptimized to remove the inserted check and prefetch instructions, and control returns to the profiling phase. For long-running programs, this profile, analyze and optimize, and hibernate cycle will repeat multiple times. Experimental
results demonstrate that the prefetching technique is effective, providing overall execution time improvements of 5–19% for several memory-performance-limited SPECint2000 benchmarks running their largest (ref) inputs.

Notice that, load with stride pattern will generate temporal data references without repeating patterns. For example, a load starting at address x with a stride 1 will generate a long temporal data references x, x+1, x+2, ..., and so on without a repeating pattern. Therefore, hot data stream prefetching and stride prefetching usually prefetch different sets of loads and they can complement each other.

5.3. Mississippi delta prefetching

Mississippi Delta (MS Delta) is a novel technique for prefetching linked data structures that closely integrates the hardware performance monitoring unit (PMU), garbage collector’s global knowledge of heap and object layout, and JIT compiler analysis. The garbage collector uses the PMU’s data cache miss information to first identify cache-miss intensive traversal paths through linked data structures, and then to discover regular distances between objects along these linked data structures. Coupling this information with JIT compiler analysis, MS Delta materializes prefetch targets in a timely fashion and injects prefetches that improve memory subsystem performance.

Figure 4 shows the high-level flow of the system as it abstracts from raw PMU samples up to prefetches. First, the IPF PMU hardware provides samples of high latency cache miss. Each sample includes the instruction pointer address (IP) and the reference effective address (EA) of the memory access. MS Delta abstracts these raw samples first into the objects that caused the miss and then into a high level metadata graph, whose nodes represent object types annotated with instruction addresses, and whose edges represent relations induced by fields and array elements containing references. During heap traversal, the garbage collector uses object samples to discover edges in the metadata graph that approximate the high latency traversals between linked data. It then composes these edges into paths representing linked data structure traversals that cause high latency cache misses. Taking advantage of object placement, the garbage collector determines regular deltas between objects along the paths. Knowing the deltas and the paths simplifies the JIT compiler analysis needed to schedule prefetches along a traversal path: The JIT combines the address of the first object in the path with the deltas to materialize prefetch targets. This means that the miss latency experienced by the first object in a traversal path hides the miss latency of subsequent objects along the path. MS Delta prefetching achieves a speedup of 12% to SPEC JBB2000 benchmark over a baseline system that performs dynamic profile-guided optimizations but no data prefetching.
6. Profile-based thread level speculations

Thread level speculation explores parallelism beyond the instruction level. It may speculatively execute multiple loop iterations or multiple regions in parallel. Java Runtime Parallelizing Machine speculate loop iterations within a dynamic compilation system. Speculative Parallel Threading speculates loop iterations in a two-pass static compiler. Computation reuse speculates regions of code. A helper thread may execute code covering multiple functions, trying to overlap cache miss latency with useful computations. In all these techniques, profile significantly improves the effectiveness of the speculations.

6.1. Java runtime parallelizing machine

Java runtime parallelizing machine (JRPM) [Chen03] is a dynamic compilation system supported by a special hardware profiler, see Section 3.4. The system is based on a chip multiprocessor (CMP) with thread-level speculation (TLS) support. CMPs have low sharing and communication costs relative to traditional multiprocessors, and thread-level speculation (TLS) simplifies program parallelization by allowing us to parallelize optimistically without violating correct sequential program behavior.

When the input (e.g. bytecode) is first translated, loops without obvious serializing dependency are annotated so the hardware profiler will analyze the speculative buffer requirements and inter-thread dependencies of the prospective speculative thread loops (STLs) in real-time to identify the best loops to parallelize. When sufficient data has been collected for a potential STL (e.g. at least 1000 iterations have been executed), the estimated speedup for a STL is computed using the dependency arc frequency, thread
sizes, critical arc length, overflow frequency and speculative overheads. Only loops with predicted speedups > 1.2 are recompiled into speculative threads. Experimental results demonstrate that Jrpm can exploit thread-level parallelism with minimal effort from the programmer. On four processors, it achieved speedups of 3 to 4 for floating point applications, 2 to 3 on multimedia applications, and between 1.5 and 2.5 on integer applications.

6.2. Speculative parallel threading

Using profile to select profitable loops for speculative parallel threading (SPT) can also be explored in a static compiler with the support of profiling. [Du04] proposed a cost-driven compilation framework for speculative parallelization of sequential programs.

The cost-driven compilation framework uses a two-pass compilation process to select and transform all and only good SPT loops in a program. In the first pass compilation, the initial loop selection selects all loops that meet simple selection criteria (such as the loop body size requirements) as SPT loop candidates. Loop preprocessing such as loop unrolling and privatization is then applied to transform the loops into better forms for speculative parallelization. For each loop candidate, the SPT compilation framework core is invoked to determine its best SPT loop partition. The result of the first pass is a list of SPT loop candidates with their optimal SPT loop partition result. All loop transformations are performed in a tentative manner and do not alter the compiled program. No actual SPT code is generated. This pass allows the compiler to measure the amount of speculative parallelism in all loop candidates (including each nested level of a loop nest) and their potential benefits/costs.

The second pass takes the result from the first pass and performs the final SPT loop selection. It evaluates all loop candidates together (not individually as in the case in the first pass) and selects only those good SPT loops. Then with the selected loops are again preprocessed and partitioned. Finally the compiler applies the SPT transformations to generate the SPT loop code.

The SPT compilation includes a data dependence profiling tool. The profiling is done offline, and the results are used during pass 1 compilation. The data dependence profile information together with the reaching probability information of the control flow graph is used to annotate the cost graph for best SPT loop partition.

The SPT compilation also uses a technique called software value prediction to reduce misspeculation overhead. For critical dependences that cause unacceptably high misspeculation cost, the compiler instrument the program (during pass 1) to profile the value patterns of the corresponding variables. If the values are found to be predictable, and both the corresponding value-prediction overhead and the misspeculation cost are acceptably low, the compiler inserts the appropriate software value prediction code to generate the predicted value in the selected SPT loop code. It also generates the software check and recovery code to detect and correct potential value mis-prediction.

With the sophisticated cost-driven compilation and profiling support, this technique may achieve an
average 15% speedup for Spec2000Int benchmarks.

6.3. Speculative computation reuse

Computation reuse [Connor99] caches the inputs and results of previous execution instances of a region of computation. When the same computation is encountered later with previously seen inputs, the cached results are retrieved and used directly without actually executing the computation. Performance of programs improves due to skipping the execution of the reusable computation.

Computation uses a buffer (called the computation reuse buffer or CRB) to cache the inputs and output for each computation region. If the inputs or outputs changes frequently, CRB will overflow and the computation may not be able to be reused. For example, if the input parameter to the region changes with a constant stride, such as $x+1$, $x+2$, ..., $x+i$, it will be unlikely that the current input matches any of the previously seen input. Furthermore the compiler needs to insert an invalidation instruction after each store instruction that may access the same address as a memory operation (either a load or a store) inside a computation region. When an invalidation instruction is executed, the hardware marks the CRB entry for computation region specified in the invalidation instruction as invalid. An invalid region will not be reused. However, the invalidation instructions often have to be inserted conservatively as alias analysis is a known hard problem. In addition, even when a store accesses the same memory location as that by a memory operation in a computation region, the stored value may remain unchanged and the region.

Speculative computation reuse [Wu-01] uses value profile to help computation reuse. It includes a loop or an instruction into a region only if the loop or the instruction has up to 5 result values that account more than 60% of all possible result values for the loop or the instruction. To prevent the unnecessary invalidation, speculative computation reuse takes advantage of the speculative multithreading hardware. In speculative reuse, there is no need to insert invalidation instructions. Instead, the output in the matching instance can be “predicted” to be correct and is speculatively reused, even though some store executed outside the region may have accessed the same address as a memory operation inside the region. The speculative multithreading hardware is used to overlap the speculative execution of the code after the region with the execution of the computation region that verifies the predicted output. The speculative reuse will not commit result to memory or register file until the verification thread confirms that the predicted output is correct. If the verification thread finds out that the reused output is incorrect, the speculative execution will be squashed. At that time, the output of the computation region is available from the verification thread and can be used in the code after the region non-speculatively.

Experimental result shows that speculative computation reuse with 8 inputs/outputs per region performs 11% using value profile better than that without using the value profile.

6.4. Software-Based Speculative Precomputation

Software-based speculative precomputation [Liao-02] is a profile-guided technique that aims at improving the latency of single-threaded applications by leveraging multithreading resources to perform memory
prefetching via speculative prefetch threads. SSP first collects cache profiles from the simulator or PMU to identify the top delinquent loads that contribute to at least 90% of the cache misses. It then analyzes the single threaded application to generate prefetch threads and identify and embed trigger points for the threads. The execution of the resultant program spawns the speculative prefetch threads, which are executed concurrently with the main thread. Experiment with Itanium-like in-order processor shows that for some pointer-intensive benchmarks, the prefetching performed by the speculative threads achieves an average of 87% speedup.

7. Issues with profile-based speculation

Although profile is effective to improve speculation accuracy and is widely used, several issues with profile-based speculation remain to be solved or need further investigations.

7.1. Stability across multiple workloads.

Profile-based optimizations are by nature speculative. Its usefulness depends on how accurate the profile collected from early run captures the future behavior of the program. Block and edge profiles are relatively stable across different workloads [Fisher92]. However, lower level profiles, such as cache profile, are more sensitive to workloads and the underlying machines [Hsu02].

7.2. Update when program changes

To support profile-based optimization, a data base of profile for the input program is stored. Large scale commercial program involves millions of lines of code and many third party modules. Even a small change in one part of the program may affect the profile globally. Recompile the whole program to regenerate the profile could be overwhelming. Program analysis may help to derive new profile for the change portion and update the profile for the code affected by the change. If this is not possible, program analysis may help identify the portion of the program to re-profile and recompile.

7.3. Maintenance during optimizations

A profile is usually used by multiple optimizations. For example, an edge profile may be used in both the function inlining optimization to decide the profitable call sites, and in region formation to expand optimization scope. However, when one optimization changes the control flow graph, the edge profile may change and the optimization must maintain the profile. Otherwise the profile may become outdated quickly.

In most cases, the profile is collected in a compilation phase before the optimizations so the profile can be used by the optimization phases. In these cases, each optimization needs to update the profile to reflect the program changes in the optimized code. This profile maintenance issue is studied in [Wu92].

If the profile is collected after the optimizations, such as the profile collected from performance monitoring, the binary level profile first needs to be mapped to the intermediate representation of the IR before it is
used by the optimization and then needs to be updated when optimizations are performed. This mapping usually requires compiler annotation about how the binary code is optimized and generated.

8. Summary

Modern processors with speculation support can benefit greatly from profile information and profile-guided optimizations. The commonly used profiles include the control-flow profile, memory profile, and value profile. Most of the commercial compilers support static analysis to obtain an estimate of the control flow profile. Program instrumentation sometimes is used collect more accurate profile with a training run. Hardware performance monitoring enables sampling based profiling without modifying the user programs. There are also special profiling hardwares in systems where normal profiling technique is insufficient or too slow. A recently trend is to use software and hardware collaboration to achieve accurate profile at low cost. Profiles can be used at compile-time to boost program performance. Dynamic optimizers typically use a two-phase approach to identify and optimize frequently executed code. For program with phase changes, continuous profiling and optimizations may be necessary to achieve the best performance.

Compilers have been very successful at improving instruction level parallelism with profile-guided optimizations. Profile-guided superblock optimization is widely used in both static and dynamic compilation systems. Code and data layout, hot-cold optimization all rely on accurate profile, and have shown good performance potentials. Recent progress demonstrates outstanding performance improvement with profile guded data prefetching.

Thread-level speculation can benefit profile information even more as mis-speculation overhead increases with the increase in speculation granularity. Profile information allows the compiler to avoid speculation that is not profitable. Profile information is also instrumental in new thread-level speculation paradigm, such as speculative computation reuse and speculative helper threads.

There are, however, many issues remaining to be resolved for profile-based speculation. The stability of a profile across multiple workloads is still unresolved. Profile-guided optimizations remain cumbersome to apply and profile update when program changes still haunts the users of profile guided optimizations. Using profile information within a compiler also pose new challenges to compiler writers as maintaining profile accurately during optimization is both subtle and error-prone.

9. References


2 A. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, S. Subramoney, “Prefetch Injection Based on Hardware Monitoring and Object Metadata,” PLDI04, June 2004


4 Annavaram, Murali, Jignesh M. Patel, Edward S. Davidson, “Call graph prefetching for database applications,”
6 Ball, T.; Larus, J.R.; “Optimally profiling and tracing programs,” ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16 Issue 4
8 Page(s): 46 -57
22 Du, Zhao-Hui, Chu-cheow Lim, Xiaofeng Li, Chen Yang, Qingyu Zhao, Tin-fook, Ngai, “A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs,” PLDI04, June 2004
24 Ebcioglu, Kemal; Erik R. Altman, Michael Gschwind, and Sumedh Sathaye, "Dynamic Binary Translation and Optimization," IEEE TRANSACTIONS ON COMPUTERS, VOL. 50, NO. 6, JUNE 2001

Sherwood, Timoth; Suleyman Sair, and Brad Calder, “Phase Tracking and Prediction” ISCA-30, June 2003.


