ABSTRACT
The performance of modern processors is increasingly dependent on their ability to execute multiple instructions per cycle. Explicitly Parallel Instruction Computing (EPIC) architectures can achieve high performance by using the compiler to express program instruction level parallelism (ILP) directly to the hardware. The predicated execution feature is critical to the success of the EPIC architecture approach because it allows the compiler to explicitly overlap the execution of independent control paths. An advantage of predicated execution is the elimination of hard-to-predict branches by executing both paths of a branch in a single code sequence. However, there are a number of advantages to predicated execution support other than simply reducing branch mispredictions and enabling the parallel execution of multiple control flow paths. To date, very little research has been performed to investigate radical changes to a program’s original data flow and control flow. This paper outlines methods of using predicated execution to substantially improve the efficiency of EPIC memory accesses. Value locality, the repetition of computation values serve as a key attribute to reduce unnecessary memory operations of a program.

1. INTRODUCTION
Memory access penalties have become a major issue for the microprocessor industry to deliver higher performance microprocessors. The growing disparity between processor and memory performance will make cache misses increasingly expensive. Additionally, data caches are not always used efficiently, resulting in large numbers of data cache misses. In numeric programs there are several known compiler techniques for optimizing data cache performance [13]. However, integer (non-numeric) programs often have irregular access patterns that are more difficult for the compiler to optimize.

As memory latencies increase, the importance of cache performance improvements at each level of the memory hierarchy will continue to grow. Rather than strictly trying to improve memory system efficiency with adaptive cache management techniques [9] it makes sense to direct advances in compiler technology to determine whether memory accesses are absolutely necessary to determine the results of program computations. By eliminating unnecessary memory accesses, it is possible to improve memory access efficiency and deal with long memory latencies, utilizing predicated execution to eliminate unnecessary memory requests. Specifically, the goal is to increase data cache effectiveness for integer programs.

Generally, optimizing compilers are successful at eliminating inefficiencies and redundancies within programs by performing iterations of analysis and optimization. The elimination of redundancies in programs at compile time can dramatically improve a program’s execution time. Traditional compiler techniques such as constant propagation, common subexpression elimination, loop invariant code removal, conditional branch elimination, and partial redundancy elimination [5] eliminate program redundancy and improve the efficiency of a program. However, compiler optimization techniques rely on the detection of static redundancy, which requires the complete assertion that the computations be definitely redundant for all executions. However, compiler techniques have limited mechanisms for exploiting dynamic values in programs. Several empirical program behavior studies indicate that many instruction traces are dynamically executed with the same inputs, a form of redundancy known as value locality [6] Overall, in order to exploit value locality for the purpose of improving the efficiency of the memory system, hardware support must be available to a processor to selectively disable memory requests that are determined as unnecessary. The EPIC architecture predication execution mechanism is a perfect match for this requirement.

Predicated execution, or simply predication, is a mechanism that supports conditional execution of individual operations based on a Boolean guard or predicate [8][16]. Predicated operations are fetched regardless of their predicate value. Operations whose predicate is TRUE are executed normally. Conversely, operations whose predicate is FALSE are nullified, and thus are prevented from modifying the processor state. With predication, the representation of programmatic control flow is inherently changed. A processor with predication can support conditional execution either via conventional branches or with conditional operations. As a result,
the compiler has the opportunity to physically restructure the program control flow into a more efficient and parallel form for execution on an ILP processor. It is the effective integration of predication with the EPIC execution microarchitecture that provides an efficient method of disabling memory instruction requests.

In [11], the benefit of predication support was studied with a predication compiler. However, the accompanying compilation model was based solely on synthesizing predicated code based on the original control flow. One fundamental problem with using traditional compiler techniques to deploy predicated execution is that existing compiler techniques do not significantly alter the program's original control and data flow structure to sufficiently use the full potential of the predicated execution mechanism. For modern compilers to expose sufficient amounts of instruction level parallelism, techniques for analyzing the original program control flow and data flow into new EPIC-aware forms must be developed. This paper illustrates new compiler transformations that exploit value locality characteristics and commonly occurring data dependence program structures to create more memory efficient code. Such execution paths can be strategically synthesized to increase the amount of instruction level parallelism that can be efficiently executed on a predicated architecture. Overall, this research proposes new compiler transformations whose projected impact is to extend the benefits of predicated execution in the EPIC architectures.

2. MOTIVATION AND BACKGROUND

Currently memory performance issues are limiting the performance of EPIC architecture implementations. It is estimated that nearly 50% of the Intel/HP Itanium execution time on the SPEC benchmark suite is spent stalled waiting for both instruction and data cache misses. Figure 1 illustrates the data cache hit rate and the percentage of cache-miss stall cycles relative to the overall execution for a few SPEC benchmarks evaluated on an EPIC architecture. It is important to note that although the benchmarks achieve nearly an average of 90% hit rate, 40% of EPIC execution is dominated by memory system stalls of the process. Although out-of-order processor systems can adapt to cache misses by executing other instructions surrounding the load, this is not the case for EPIC machines. For an EPIC in-order design, the execution pipeline stalls when another instruction attempts to use the result of a load that has missed in the cache. As this execution semantic is a substantial performance bottleneck, it is critical to improve memory efficiency by eliminating unnecessary demands on memory system. The inherent problem requires substantial analysis in synthesizing code with more efficient memory access patterns and directing new compiler-directed prefetch mechanisms. To improve memory system efficiency, the compiler will need to play a more active role in analyzing and transforming the original data flow structure of a program.

There are a number of advantages to an EPIC architecture's predicated execution support than simply reducing branch mispredictions and enabling the parallel execution of multiple control flow paths. To date, very little research has been performed to investigate radical changes to a program's original data flow and control flow. This paper outlines methods of using predicated execution to substantially improve the

Itanium’s memory access efficiency. Value locality, the repetition of computation values may serve to reduce necessary memory operations of a program. Although new architecture attempts have been proposed to simply reuse previous results of execution [18] based on value locality, there are more immediate benefits that can be made in the Itanium architecture using predicated execution.

By analyzing data dependence graphs with value locality profile information, several dependence graph structures can be transformed to eliminate the number of necessary memory accesses. In a dependence graph there are often disjunctive and conjunctive functions applied at the join point of two sequences of dependent operations. Consider a dependence graph with two halves, section A and section B. Certain values generated by section A of the graph will render the operations on the section B unnecessary. If load operations are present on section B of the dependence graph, they also are unnecessary. With the in-order nature of an EPIC architecture, the section B loads can be predicated based on finding the run-time value in the first dependence chain. In the case that newly predicated loads would miss in the cache, their predicate operands will indicate that it is unnecessary to access the memory location.

As a second example, consider the simple set of integer operations displayed in Figure 2 that may be found in a loop structure. In this case, it is not necessary to perform the loads of array elements B[i] and C[i], if the result of A[i] is zero. In this case, zero multiplied by anything will remain zero. By predicating loads to array B and C are predicated based on the respective value of array A, many unnecessary cache misses are eliminated. Similarly, silent store (store operations writing the same value to memory) can also be eliminated. Overall, there are two beneficial effects of eliminating unnecessary memory accesses: elimination of memory latency and elimination of cache pollution. As such, the proposed approach may have substantial effects in aiding
\[ X[i] = A[i] \times (B[i] + C[i]) \]

(a)

Figure 2: Code example of value-irrelevant memory accesses (a) source code and (b) dependence graph.

the Itanium architecture’s performance.

Figure 3 illustrates the degree to which zero valued loads contribute to cache inefficiency. Cache simulations for this set of SPEC Benchmarks revealed that load operations which evaluate to zero occur quite frequently. Furthermore a significant set of these loads are responsible for data cache misses. Since the occurrence of zero valued loads is relatively high, it appears that cache efficiency could be increased by applying value locality optimization techniques. Based upon this information, our proposed work is to use predicate instructions to eliminate the execution of memory operations which could be removed due to their data dependence on zero valued functions.

According to the IMPACT EPIC cache simulations which were run using the IMPACT compiler suite, load operations that evaluated to 0 for more than 95% of their executions, contributed as much as 20% to the total data cache misses. Although the distribution of the zero valued load operations are varied over the benchmarks, the presence of such high percentages beckons the use of our proposed optimizations. 132.ijpeg and 181.mcf appear to be the only benchmarks that did not exhibit a bias toward zero valued loads. The others showed significant contributions to data cache misses from these operations.

2.1 Background and Related Work

Related Work. There has been significant previous research work in the area of program reformulation. In addition to the arithmetic reformulation work mentioned in the previous section, a large body of related work in the area of control flow optimization has also been carried out. These methods can be classified into three major categories: branch elimination, branch reordering, and control height reduction. Branch elimination techniques identify and remove those branches whose direction is known at compile time. The simplest form of branch elimination is loop unrolling, in which instances of back-edge branches are removed by replicating the body of the loop. More sophisticated techniques examine program control flow and data flow simultaneously to identify correlations among branches [4], [14]. When a correlation is detected, a branch direction is determinable by the compiler along one or more paths, and the branch can be eliminated. In [14], an algorithm is developed to identify correlations and to perform the necessary code replication to remove branches within a local scope. This approach is generalized and extended to the program-level scope in [4]. The second category of control flow optimization work is branch reordering.

Another category of program reformulation is control flow optimization research, which focus on the reduction of control dependence height. This work attempts to collapse the sequential evaluation of linear chains of branches in order to reduce the height of program critical paths [17]. In an approach analogous to a carry look-ahead adder, a look-ahead branch is used to calculate the taken condition of a series of branches in a parallel form. Subsequent operations dependent on any of the branches in the series need only to wait for the look-ahead branch to complete. The control dependence height of the branch series is thus reduced to that of a single branch. The mechanisms introduced herein also serve to reduce control dependence height. This chapter, however, leverages an approach [3] to minimization and re-expression of control flow networks that is far more general than those proposed in previous work.

The final category of related work is value-based compiler transformations. Value-based optimization include all optimization based on a predictable value or range of values for a variable or instruction at run time. These include constant propagation, code specialization [7], optimization assuming the value predictability of an instruction, continuous optimization, and partial evaluation. Variables and instructions that have invariant or predictable values at run time, but cannot be identified as such using compiler analysis, can benefit from value-based compiler optimization.
3. APPROACH
Motivated by the potential of aggressive techniques for transforming arithmetic expressions, this section introduces a new approach to optimizing program structures. The goal of this work is to develop a systematic methodology for reformulating specific regions of program data flow for more efficient exploitation of their inherent value locality characteristics on an ILP processor. Dynamic value locality information is proposed to be gathered and then represented as a dynamic value dependence graph. A new, more parallel computation is synthesized with the goal of reducing dependence height. To accomplish the desired optimization and synthesis, the parallel computation is modeled as new control flow structures which are added through a proposed process known as if-insertion. In turn the control structures are converted to a predicate representation using the traditional if-conversion technique.

Figure 5 illustrates the program transformation paths and compiler techniques presented in this section. There are two illustrated program representation paths: traditional control flow (CF), to predicate data flow (PF) using if-conversion, and traditional control flow (CF) to if-inserted value-flow (VF) to predicate data flow (PF). The predicate representation of code effectively designates the execution conditions of the newly created program sequences. The key idea behind the value-based optimization is to synthesize optimized code structures by analyzing the dynamic profile-determined program value characteristics and applying a series of control flow and data flow transformations that use both a predicate-based compiler representation and predicated execution microarchitecture support.

Two domains of transformation techniques are proposed: predicate-based data-irrelevant and control-irrelevant memory elimination transformations. The transformations are guided with value profiling [6] and cache profiling information gathered during training runs of selected benchmarks. The profiling information is used to generate new control instructions within the code (if-insertion) which are then

Figure 4 contains a simple example illustrating the concept of predicated execution. Figure 4(a) shows sequential if-then-else constructs, called hammocks. The branch outcomes are determined by the evaluation of the branch condition Cond1. Depending on the outcome of the first branch register, r1 is either incremented or decremented. Figure 4(b) shows the code segment after if-conversion, and Figure 4(c) shows the code after scheduling. Here the two branch conditions have been transformed into comparison instructions that define predicate destinations. The example illustrates how support for predicated execution allows the multiple path contexts to be executed in parallel on a wide-issue machine.

Figure 4: An example code segment (a), after if-conversion (b), and after scheduling (c).

Predicated Execution Background.

Predicated execution is a mechanism that facilitates the conditional execution of individual instructions. Predication is most commonly utilized in a compiler by employing if-conversion. If-conversion is a process whereby conditional branches are converted into predicate defining operations, and operations along alternative paths of each branch are guarded under the computed predicates [1][15][12]. Predicates are registers that store a single bit value, representing either TRUE or FALSE. Each instruction is associated with a particular predicate, known as its guard predicate, that determines its execution.

With if-conversion, complex nets of branching code can be replaced by a straight-line sequence of predicated code. There are two major benefits associated with applying if-conversion. First, a compiler can eliminate problematic branches from the program. In doing so, all the associated overhead with these branches is removed, including misprediction penalties, penalties for redirecting sequential instruction fetch, and branch resource contention [10]. Second, predication facilitates increased ILP by allowing separate control flow paths to be overlapped and simultaneously executed in a single thread of control.
collapsed into predicate-based data dependent instruction using if-conversion.

Eliminating Data-Irrelevant Memory Accesses.

The data dependence graph of a particular computation may compute the same result by only executing a subset of the graph’s execution nodes. For example, Figure 6 illustrates an example from 126.gcc of memory operations that become irrelevant to code’s result when some instructions resolve to the value zero. The source code of Figure 6(a) updates two arrays (basic_block_live_at_start and basic_block_live_at_end) being updated based on the result of a logical-or-and sequence with their original values. Specifically this code is performing dataflow analysis within a compiler and calculating the live range of a set of program variables. The loop is invokd several times, yet the live variable information is very sparse (zero-valued loads) and change infrequently. The execution is unnecessary when the logical-and operation result is zero and any cache misses are unnecessary.

Eliminating Control-Irrelevant Memory Accesses.

Figure 7 shows an example of Control-Irrelevant memory guarding. Figure 6(a) shows a code block with a branch followed by a load-add pair. Speculative placement of the load moves the load above the branch. Because the add is dependent upon the load, the add can also be speculated above the branch in Figure 7(b). Speculating the load allows for increased parallelism through code reordering. The problem with speculation is that the load is executed every time that particular section of code is entered, regardless of whether or not the branch takes. Because the load is irrelevant if the branch takes, it is Control-Irrelevant. There are two main problems with Control-Irrelevant loads. The first is the cache pollution caused by loading a value that isn’t used or needed. The second is important for in-order machines. If the speculative load misses in the cache the machine must stall on the speculative add instruction. If the branch takes the machine stalled for no reason. This is an obvious performance deterrent that must be addressed. We propose a predicate based method called Control-Irrelevant Memory Guarding. For the case in Figure 7 at the time the load is executing the condition determining whether or not the branch is taken is known. Using predication, we can predicate the speculative instructions so that they only execute if the branch is not taken Figure 7(c). By using predication to guard the speculative instructions, we can achieve the benefits of speculation without the drawbacks of irrelevant execution.
4. EXPERIMENTAL EVALUATION

4.1 Methodology

The IMPACT compiler and emulation-driven simulator were enhanced to support a model of an IMPACT EPIC architecture [2] and simulation of the code transformation techniques respectively introduced in Section 3. The benchmarks used in all experiments consist of SPECINT95 and SPECINT2000 programs. The base level of code consists of the best code generated by the IMPACT compiler, employing function inlining, superblock formation, and loop unrolling.

The base processor modeled can issue in-order six operations up to the limit of the available functional units: four integer ALU’s, two memory ports, two floating point ALU’s, and one branch unit. The instruction latencies used match the Itanium microprocessor (integer operations have 1-cycle latency, and load operations have 2-cycle latency.) The execution time for each benchmark was obtained using detailed cycle-level simulation. The parameters for the processor include separate 32K direct-mapped instruction and data caches with 32-byte cache lines with a miss penalty of 12 cycles. A 1M second-level unified cache (with memory latency of 100 cycles) is also simulated. For branch prediction, a 4K entry BTB with two-level correlation prediction with a branch misprediction penalty of eight cycles is modeled.

4.2 Results and Analysis

Figure 8 illustrates the percentage of cache missing loads that are caused by a speculated memory operation. On average, approximately 44% of loads that miss in the cache are speculative. Preliminary analysis shows that approximately 1/3 of the speculated loads prove to be control irrelevant. This means that 1/3 of the loads in Figure 8 do not need to be executed. Speculative loads allow for increased parallelism, however, this data shows that speculation can also have negative effects in the form of irrelevant execution. Irrelevant execution of loads causes cache pollution and possible stalls for speculated data that is not needed. This indicates a high potential for optimization. By using predication to guard speculative loads from control irrelevance, the benefits of speculation can be achieved without the penalty of irrelevant execution.

Figure 9 displays the opportunity for removal of control irrelevant loads via predication. Of the speculative, control irrelevant loads that cause a miss, Figure 9 displays the percentage that could be removed through the proposed predication techniques. Compiler analysis indicates that on average 60% of irrelevant speculative loads could be guarded against irrelevant execution. This indicates a large number of candidates for predicate based optimization.

The performance speedups of Figure 10 indicate that all benchmarks tested improved markedly with the proposed compiler based predicate transformations. The first data set compares the data irrelevant code to the base case of superscalar optimized code. It is noteworthy that the data dependence removal proves beneficial on code that has already passed an aggressively optimizing compiler. Similarly, the results from the second data set demonstrate that the removal of data dependence still boosts performance when run with the classic control removal predicate instructions.

Benchmark 124.m88ksim exhibited the greatest speed up, 6% for both cases. This was expected due to the high percentage of zero valued load instructions, 31%, shown in Figure 3. While 130.li only improved by 3%, which again reflects a smaller percentage of zero valued loads during execution, 14%. Cache performance evaluations indicate that...
the performance is achieved by eliminating between 3-5% of the original cache misses.

5. SUMMARY
Current advanced compilation techniques do little to alter a program’s original control flow. Because of this, modern optimization can do little to reduce the redundancies and irrelevant execution so common in today’s programs. Through the use of profile guided predication, new methods were developed to alter the inherent control flow of a routine, resulting in increased performance and elimination of redundant execution. By changing the data and control flow through predication, improved memory access efficiency is achieved. Because current predicated architectures are implemented on in-order machines, the memory system’s performance is vital. Using predication to remove data and control irrelevant loads results in a more pollution-free memory system as well as performance benefits resulting from the elimination of unnecessary stalls. Preliminary experiments show increased performance with relatively simple predication optimization.

The techniques presented in this paper are confined to the scope of compilation trace of code. Although interesting results were established with the system, future work will investigate methods of determining whether many other memory accesses are necessary to a program’s execution. Such a technique will require a global analysis and optimization infrastructure.

6. REFERENCES


