To Appear in *Proceedings of the 32nd Annual International Symposium on Microarchitecture* (MICRO), November 1999.

Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results

Daniel A. Connors and Wen-mei W. Hwu
Department of Electrical and Computer Engineering
Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801
Email: {dconnors, hwu}@crhc.uiuc.edu

Abstract

Recent studies on value locality reveal that many instructions are frequently executed with a small variety of inputs. This paper proposes an approach that integrates architecture and compiler techniques to exploit value locality for large regions of code. The approach strives to eliminate redundant processor execution created by both instruction-level input repetition and recurrence of input data within high-level computations. In this approach, the compiler performs analysis to identify code regions whose computation can be reused during dynamic execution. The instruction set architecture provides a simple interface for the compiler to communicate the scope of each reuse region and its live-out register information to the hardware. During run time, the execution results of these reusable computation regions are recorded into hardware buffers for potential reuse. Each reuse can eliminate the execution of a large number of dynamic instructions. Furthermore, the actions needed to update the live-out registers can be performed at a higher degree of parallelism than the original code, breaking intrinsic dataflow dependence constraints. Initial results show that the compiler analysis can indeed identify large reuse regions. Overall, the approach can improve the performance of a 6-issue microarchitecture by an average of 30% for a collection of SPEC and integer benchmarks.

1. Introduction

One of the major challenges to increasing processor performance is overcoming the fundamental dataflow limitation imposed by data dependences. By reusing previous computation results, the dataflow limit can be surpassed for sequences of operations that are otherwise redundantly executed. Traditional compiler techniques for eliminating program redundancy include common subexpression elimination, loop invariant code removal, and partial redundancy elimination [1]. These optimization techniques rely on the detection of static redundancy, which requires the computations be definitely redundant for all executions. As such, compiler techniques have no mechanism for capturing dynamic redundancy which occurs over a temporal set of definitions. As a result, several empirical studies indicate that significant amounts of redundancy, or *value locality*, still exist in optimized programs [12][14][17].

To exploit dynamic redundancy, two hardware strategies, speculative value prediction [11] and dynamic instruction reuse [16], have been proposed. Due to hardware complexity limitations, these techniques detect reuse opportunities at the instruction level rather than at a larger granularity. A more aggressive alternative is to allow the compiler to partition the program into potentially reusable regions of computation whose results are then dynamically recorded in hardware for future reuse.

Consider the loop example in Figure 1, which computes the sum of the elements in the array A. To improve program execution speed, it is desirable to remove the loop's computation when the resulting sum is identically computed with a previous invocation. Assume that the loop is first invoked at a time τ and then at a later time $\tau + \delta$. Additionally assume that the loop is not located within a program domain in which the compiler could trivially detect the opportunity to avoid re-computation of the sum. As such, the reuse of the computation is based on determining the equivalence of the array A at time $\tau + \delta$ and time τ, for which there is a previously computed sum. The equivalence holds if array A remains unchanged along all executed program paths between τ and $\tau + \delta$. Once the equivalence is established one can simply use the execution result recorded at τ to eliminate the need to execute the entire loop at $\tau + \delta$. Traditional compiler optimization techniques and run-time hardware mechanisms are incapable of exploiting dynamic redundancy in such regions.

A computation reuse mechanism can provide performance improvement if the system can significantly re-

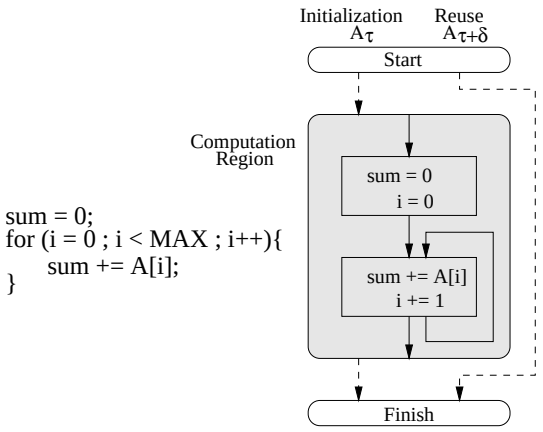


Figure 1. Loop example with potential reuse.

duce execution time for computing previous results and can adapt to run-time variations. Different trade-offs exist between hardware-based run-time methods and software approaches, and many design alternatives have been considered [13][16][18]. Our approach strives to achieve the best of both concepts in an integrated architecture and compilation framework. Toward this end, we develop instruction set architecture extensions and microarchitectural mechanisms, referred to collectively as the Compiler-directed Computation Reuse (CCR) approach. The approach allows the compiler to identify code regions whose computation can be reused during dynamic execution. The instruction set architecture provides a simple interface for the compiler to communicate the scope of each reuse region and its live-out register information to the hardware. During run time, the microarchitectural components of the approach record the execution results of the regions for potential reuse. Explicit designation of computation reuse allows a large number of dynamic control, memory, and arithmetic instructions to be removed from the processor execution. Inclusion of control instructions within the computation further increases the exploitation of value locality over block-level reuse methods [10][8].

Our current compiler implementation relies on profile-guided heuristics to identify which computations will be potentially reusable during the execution of the program. Specifically, value profiling techniques [2] enable individual instructions to be classified according to their reuse potential. Instructions with high reuse potential then serve as seeds of dataflow and constant propagation analysis to determine which regions of the program should be dynamically reused. Such regions are called *Reusable Computation Regions* (RCRs). The compiler then uses the instruction set architecture of the proposed framework to instruct the hardware how to effectively reuse the computation defined within these regions.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of works related to the concept of reuse and the intuitive rationale behind the proposed scheme. Next, Section 3 presents an overview of the architecture that facilitates the exploitation of reusable computation designated at compile time. Section 4 details the compilation issues and challenges associated with detecting region-level computation reuse. The effectiveness of the proposed approach in improving performance and exploiting instruction repetition is presented in Section 5. Finally, the paper is summarized in Section 6.

2. Related Work and Motivation

2.1. Related Work

Several empirical studies indicate the presence of significant amounts of dynamic redundancy in programs [12][14][17]. Previous research in the area of value locality and redundancy exploitation can be classified into three major categories: value prediction, dynamic instruction reuse, and memoization. Value prediction and dynamic instruction reuse are two important hardware strategies that attempt to reduce the execution time of programs by alleviating the dataflow constraints at the instruction level. Value prediction [11] speculates the results of instructions based on previous execution results, performs speculative computation using the predicted values, and confirms the speculation. Instruction reuse [16] recognizes that many instructions have the same inputs when executed dynamically, and that by buffering the previous results, future dynamic instances can avoid execution by simply using the saved result. Although alternative schemes include *dependence chains* of multiple instructions or use profiling information to guide the detection mechanism [6], the performance improvement of these proposed approachs is often limited by the exploitation of value locality at the instruction level [18]. In the *block* [10] and *trace-level reuse* [8] techniques, hardware mechanisms are proposed to exploit value locality for large straight-line sequences of instructions. These approaches detect that the inputs and outputs of a chain of instructions are highly correlated, and recognize that the inherent benefits of prediction and reuse only materialize when a large amount of execution is eliminated.

The final category of value locality exploitation research focuses on memoization. Generally, memoization is a technique that stores previous results of computation in memory, and later invocations are preceded by table lookups for already computed results. Functional and logic programs use software concepts of memoization, whereas the *Tree Machine (TM)* [9] and *result cache* [13] are hardware implementations of computation memoization. In these models, computation caching exploits value locality in the way that cache memory systems exploit spatial and temporal locality of memory accesses.

2.2. Motivation

This section presents the intuitive rationale behind compiler-directed computation reuse.

2.2.1 Opportunities for Compiler-Directed Reuse

The goal of any computation reuse scheme is to minimize the execution time of computing results that have been previously determined. The following examples illustrate that an integrated compiler and architecture reuse approach has the potential to eliminate large sequences of dynamic instructions.

Block-level reuse. Figure 2 represents an example from the SPEC92 benchmark *008.espresso* that demonstrates the complexity of efficiently detecting sequences of reusable instructions. A macro definition for computing the number of bits set to logical 1 in a 32-bit word is shown in Figure 2(a). The macro divides the 32-bit word into four bytes and uses each byte as an integer index for the bit_count array. The four byte components are then summed together. The dependence graph for this segment is shown in Figure 2(b) and uses the following instruction key: **A** for arithmetic/logical, **L** for load, **R** for right shift, **S** for left shift, **M** for move, and **B** for branch. In this case, all the code falls into one basic block because there is no possibility of branching until the end of the instruction sequence. The dependence graph illustrates that the entire sequence of operations is dependent on a single input register $r3$ and defines a single output register $r26$. No other registers defined in the sequence are live-out, i.e. used after the computation sequence. Also, by performing alias analysis, it can be determined that the array bit_count is static and does not change during program execution. The instruction sequence clearly designates an opportunity for reusing previously computed results.

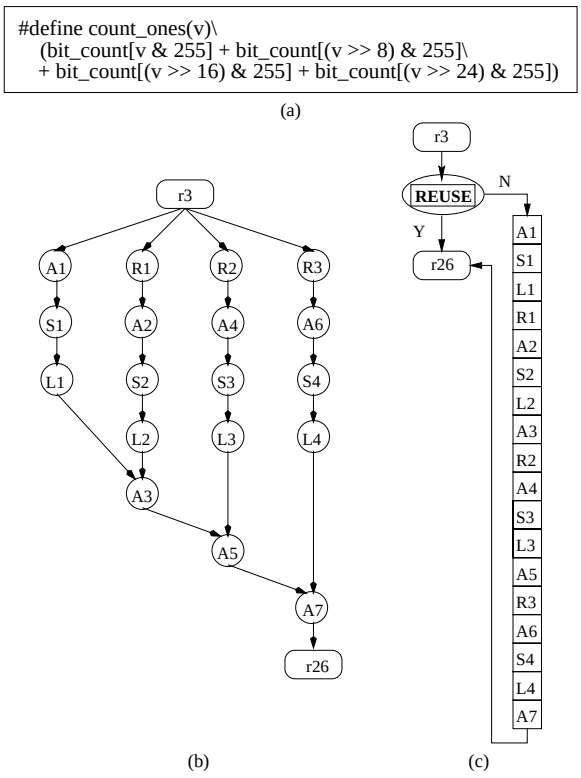


Figure 2. Block-level reuse example (a) source code macro definition, (b) dependence graph, and (c) potential reuse sequence.

The example of Figure 2 demonstrates several fundamental barriers to effective exploitation of dynamic redundancy for both hardware-based and software-based methods. First, in exploiting the full redundancy of the sequence, a run-time hardware scheme must detect the dependences between instructions as displayed by the dependence graph of Figure 2(b). The dependence representation allows the mechanism to determine the set of instructions that are reusable from a particular set of starting instructions. Similarly, the hardware approach is limited in scope and may be unable to determine that only a single register is live-out of the computation. The alternative of storing the results of all registers defined in the region can be very costly. Third, the example could also benefit from code specialization, a software scheme that duplicates the code to efficiently handle certain run-time values. Although value profiling can be used to determine whether certain variables have the same value across multiple input sets, code specializations can not easily adapt to variations in the value set.

A compiler-directed hardware approach has the advantages of both hardware and software methods. At compile time, the mapping relation between the single input register and single output register may be determined. With this information, the compiler can construct an alternative control flow graph, as illustrated in Figure 2(c), based on a reuse instruction that communicates with hardware buffers to detect reuse scenarios. If the hardware determines that a previous computation can be reused, the reuse instruction will update the corresponding registers and proceed to the next sequential instruction. Otherwise, no recorded computation can be reused and the reuse instruction will branch to the original sequence, which executes and returns. In contrast to hardware schemes, the compiler-directed approach can accurately inform the hardware of the input and output registers which, in Figure 2, are $r3$ and $r26$. Compared to software code specialization techniques, the reuse instruction can be implemented to use multiple recorded instances, allowing a large number of instructions to be skipped for several input sets.

Region-level and memory reuse. Control and memory dependences also limit the effective exploitation of dynamic redundancy. Figure 3 illustrates an example of potential reuse involving control and memory operations in the func-

tion *ckbrkpts* from the SPEC95 benchmark *124.m88ksim*. Figure 3(a) shows the source code in which the function scans the contents of the array *brktable* for breakpoint information that is updated from a set of only four functions: *nobr*, *br*, *settmpbrk*, and *rsttmpbrk*. During program execution, the code behaves as a reusable computation region since it is repeatedly executed without subsequent calls to any of the four functions that change the contents of the *brktable* array. As such, the results of one execution can be reused until the *brktable* array is changed.

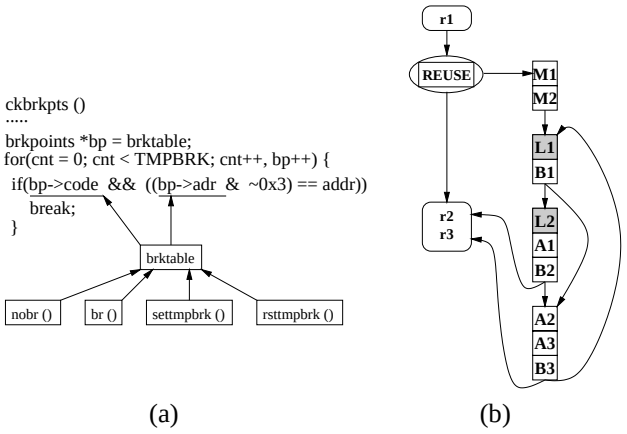


Figure 3. Region-level reuse example (a) source code and (b) reuse sequence.

Similar to the block-level example, the region example of Figure 3 presents several fundamental barriers to exploiting reuse opportunities. First, since control instructions designate program direction changes, any potential reuse detection mechanism must have the ability to understand the start and completion of the computation intended for reuse. Essentially a run-time scheme must construct the implied control flow graph of Figure 3(b) and the dependence relations among its instructions. Only by constructing such information can attempts be made to reuse execution results along separate control paths and loop regions. The detection mechanism must also identify that certain instructions within loops, such as the increment of the loop index variable and the loop-back branch in Figure 3, do not exhibit repetition of all their operands. Such instructions are integral to the operation of the loop, yet their run-time behavior may inhibit the detection mechanism in determining that the entire loop is indeed reusable. Similarly, the memory instructions of the loop are an obstacle to effective reuse. In order to reuse the results of previous loop invocations, the reuse approach must determine if array *brktable* remains unchanged along all executed program paths between an initialization time and a reuse time.

Since the hardware cost to perform complex control analysis is high, most schemes are limited to only exploiting reuse along sequential sets of instructions. However, the ability to eliminate redundancy across basic blocks is fundamental to exploiting the full potential of computation reuse. Likewise, determining the equivalence of memory structures at different times requires substantial communication between the memory system and the reuse mechanism. A compiler-directed reuse approach has the advantage of being able to exploit reuse for large regions of instructions by 1) communicating the scope of each region to the hardware responsible for storing dynamic instances and 2) communicating the equivalence of memory structures at different times of the program execution.

The communication of a region boundary designates the section of code that can be dynamically reused. The most practical region of instructions that can be easily conveyed to hardware is defined by a single starting point and a single ending point. This definition allows all control path executions between the two points to be potentially exploited by the underlying hardware reuse mechanism. A compiler-directed approach could transform the code, as illustrated in Figure 3(b), by introducing a reuse instruction to inform the hardware that a sequence could potentially be reused. The reuse of memory computations can be significantly aided by the analysis techniques employed by modern optimizing compilers. Using interprocedural analysis, the complete points-to relation [4] for the *brktable* array can be constructed at compile time. As such, the compiler-directed approach can direct the program points that affect the array to invalidate previously recorded computations based on the contents of the array. This provides potential reduction in the cost of recording computations using memory since otherwise the consistency of all 16 entries of the *brktable* array must be maintained. Otherwise, as long as the equivalence is established, future invocations can simply use the execution results recorded at an earlier time.

2.3. Computation Reuse Potential

Several studies [8][16] have determined the limits of instruction-level reuse by checking whether a dynamic instruction and its current inputs are the same as a previous execution. In this section we are interested in the potential of computation reuse occurring at a level greater than the instruction level. Reuse of a sequence of instructions is more attractive since a single reuse may eliminate the execution of a potentially long sequence of dynamic instructions. Similarly, code sequences such as the computation of Figure 1 exhibit recurrence without necessarily having repetition at the instruction level. Therefore, in examining the reuse potential of our proposed approach, we measure the amount of program execution that is redundant in the form of sequences of instructions. To do this, we constructed a value profiling infrastructure within the IMPACT compiler and emulation framework to record reuse opportunities for basic blocks and regions of code. Regions are defined as

paths of basic block segments and include both cyclic and acyclic formations. Reuse for blocks and acyclic regions is detected by considering sequences of values consumed and produced by instructions as a program executes. Store instructions were not considered to have reuse opportunities. Load instructions were considered reusable if their source memory location had not been accessed by any store operation between load executions. Reuse for cyclic regions is detected by monitoring additional program state at the invocation of the respective region headers.

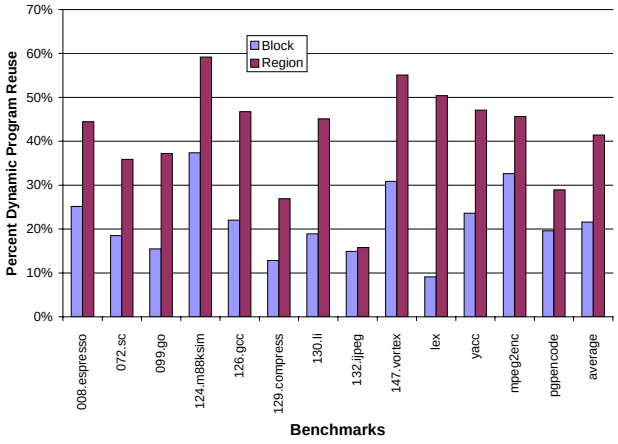


Figure 4. Dynamic reuse potential.

Figure 4 illustrates the amount of program execution satisfying the evaluation guidelines. Each figure includes a *block* and *region* column respectively indicating the amount of reuse available in basic block and region forms. For these results, eight records of previous dynamic information for each code segment were maintained to check the potential reuse of program execution. The block reuse shows an upper bound on the portion of a program that can be exploited with previously proposed techniques. On the other hand, the region-level exploitation subsumes the basic block definition and can exploit reuse along several control decisions. These results indicate that region-level reuse mechanisms can potentially exploit almost twice the amount of program execution available to block-level approaches.

3. Architecture Support

The Compiler-directed Computation Reuse (CCR) scheme extends the idea of run-time instruction reuse by introducing a set of hardware features and instructions to eliminate the need to dynamically detect reusable computation. In contrast to run-time instruction reuse schemes, the reusable computation is designated at compile time. The proposed architecture mechanism consists of the following components:

Reuse Architecture Hardware that records the dynamic computation information. The Reuse Architecture consists of a Computation Reuse Buffer (CRB) to store the reuse information.

Reuse Instruction Set Extensions Instruction extensions and execution semantics for conveying program information to the Reuse Architecture.

Computation Reuse Microarchitecture Hardware components that validate the recorded computations stored in the CRB and performs the update of architectural state for successful reuse of the computation instances.

3.1. Computation Reuse Buffer Design

To achieve the reuse goals in the compiler-directed hardware approach, a caching structure is designed. Figure 5 depicts the basic model of the structure, called the Computation Reuse Buffer (CRB). The CRB is a set-associative structure indexed by an identifier number which is specified by the proposed ISA extensions in the CCR framework. The structure is similar in design to a cache that consists of an array of entries, referred to as *computation entries*. An entry supports the reuse for a particular compiler specified region by detecting the situation in which all of the input information to the region is recurrent. To do this, each active entry is responsible for recording computation information for future region executions. As such, each entry contains four fields: 1) the computation tag; 2) a *valid* bit indicating whether the entry currently contains a valid computation; 3) an array of computation instances; and 4) a Least Recently Used (LRU) information array for managing the replacement of the computation instances. The computation tag field contains the computation identifier and is used for verifying the exact computation. A *computation instance* is defined as the set of input register operands and their respective values, the set of output register operands and their respective result values, and the validation of memory state used by the computation. A computation instance is reusable when its input register values match a previous execution of the computation and the input memory state has not been invalidated.

Multiple computation instances are used to record computations with different input values available for reuse. Each computation instance has two banks that contain an array of register entries, a *computation instance valid* bit indicating whether the instance defines a valid reuse, and a *memory valid* field indicating whether the computation accesses memory and is valid. Each register entry consists of three fields: the register index, the register value, and a *valid* field. The two banks respectively designate the necessary input and output information for the computation being reused, and the number of register entries is also specified by the particular implementation. For the input bank, the register index and register value fields record the necessary values that the respective registers must hold for the computation instance to be reusable. For the output bank, the

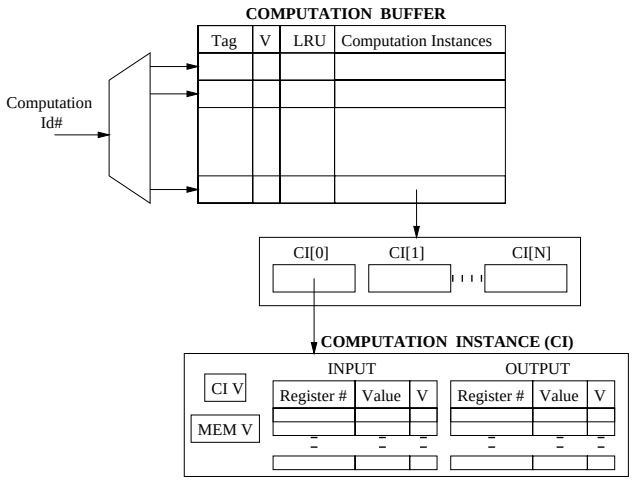


Figure 5. Computation Reuse Buffer (CRB).

information fields record the complete results of the computation that update the architectural register file during reuse. The valid fields indicate whether the register entry is active for the computation instance.

Several design enhancements can be made to the base CRB architecture. These enhancements focus on creating specific rather than uniform implementations of computation entries and instances. For instance, one enhancement might be to partition the design space of the computation entries to include variations on the type of computation instances that could be recorded. A second enhancement would include variations in the number of computation instances available for different computation entries. In Section 5 we examine the characteristics of reusable computations designated by our compiler and evaluate some variations in the design of the base CRB configuration. However, we concentrate on evaluating the initial rationale of the approach, rather than investigating these specific implementation details.

3.2. Instruction Set Extensions

The CCR approach involves the introduction of new instruction extensions and two new instructions. The new extensions designate certain aspects of reusable computation to the reuse framework. The two new instructions are: 1) *computation reuse*, which directs the hardware to determine if a computation has already been performed; and 2) *computation invalidate*, which directs the hardware to invalidate computations based on memory state changes. If the hardware does not find an opportunity to reuse previous computation results, the reuse instruction will branch to the computation code, which executes the sequence of instructions and updates the computation buffer. The reuse instruction provides a low overhead method of communicating with the hardware about the state of the machine, and is similar to the proposed mechanisms in data speculation [7] and software-controlled value prediction [5].

The introduction of the reuse opcode and instruction extensions allows the compiler to designate regions of computation that can be executed and then subsequently reused. The approach accomplishes this by a having a *region memoization* mode of execution that begins when the reuse instruction fails to find a valid computation instance. Upon starting the mode, the LRU computation instance is selected and construction of a new instance begins. Any register used before being defined while in this mode will record its information in the input bank of the instance. Additionally, instructions executed in the memoization mode have specific requirements for updating the records of the reuse instance and terminating the memoization mode. As such, the proposed ISA extensions enable the following execution semantics during the memoization mode:

Live-Out Register One new instruction extension is used to designate instructions generating live-out values. Destination registers defined for instructions marked with live-out extensions record the respective information in the output bank of the instance.

Load Instruction Load instructions executed during the memoization mode set the computation instance's memory valid flag.

Control Instruction The compiler designates the end of the memoization mode by marking certain control instructions with new extensions to indicate computation region reuse endpoints and region exits. The recording of a computation instance occurs when a reuse endpoint instruction is used to leave the region.

The execution results gathered during a successful memoization mode define a particular path in the region of instructions selected by the compiler. The compiler has the responsibility of insuring that the number of registers in the statically assigned reusable region can fit within the capacity of the computation instances. Similarly, another compiler responsibility is the program-level placement of invalidation instructions for regions accessing memory. The invalidations instruct the region computations that changes have potentially been made to the region's input data, and the computation instances may no longer be valid.

3.3. CCR Microarchitecture

The interaction of the microarchitecture pipeline and the computation reuse buffer is illustrated in Figure 6. For the initial study of the proposed approach, an in-order issue microarchitecture model is assumed, although the discussion contains relevant material applicable to a generic dynamically scheduled superscalar processor. Four unique tasks define the reuse execution for a particular computation region: accessing the CRB computation entry, reading the architectural state for the computation instances, validating

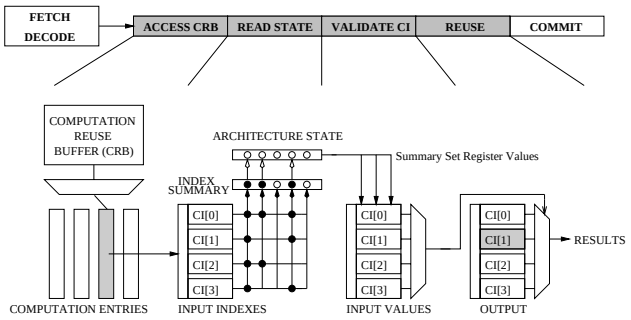


Figure 6. CCR microarchitecture pipeline.

reusable computation results, and issuing the results. If a resuable result is found, the recorded results are committed.

During the CRB access step, the CCR architecture either detects a valid computation entry for the reusable region, or flushes partially executed instructions and directs the instruction fetch stage to begin executing new instructions at the location of the resuable code segment. For valid computation entries, it is necessary to read the architectural state corresponding to the input registers of the corresponding computation instances. To do this, a summary set of the active input registers for all of the computation instances is maintained during the creation of the the instances. The summary set is the set of registers that require operand values from either the committed architectural state of the register file or uncommitted instruction results. In-flight instruction results and in-flight invalidate instructions force the reuse instruction to wait in order to validate the recorded computation instances. An interlocking mechanism exists between the computation reuse instruction, the processor retirement stage, and the register bypass circuitry to determine the necessary wait scenarios. Since the performance of the reuse scheme is determined by the reuse latency and the percentage of reused instructions, these timing considerations have been simulated within the evaluation environment.

After reading the stable computation instance input sets, the computation instances are validated. There are two possible cases that result from the validation step: the nonexistence of reusable computation and the presence of valid computation results. For successful reuse determination, the live-out registers are updated by issuing multiple results to the retirement buffer. Otherwise the processor flow of control is directed to the computation code after the pipeline is cleared of partially executed instructions.

4. Compiler Support

Our current compiler implementation relies on profiling information and dataflow analysis to direct the hardware to the regions of code that should be dynamically reused. Compiler support of the CCR approach involves four components: deterministic computation, value profiling, reusable computation regions, and reuse selection heuristics. Support for compiler-directed computation reuse is implemented in the IMPACT compiler framework. This section describes the four components related to the determination of reusable computation within general purpose imperative programming languages.

4.1. Deterministic Computation

The compilation techniques for dynamic computation reuse are based on the concept of *deterministic* computation regions. A deterministic computation region is an arbitrary, connected subgraph of the program control flow graph that can be analyzed to determine the location of all input operands that affect the region's computation. In the context of the CCR framework, two classes of deterministic regions exist: *Stateless* (SL) and *Memory Dependent* (MD). Stateless regions are simply paths of code that define computation results which are based only on register operands and not on memory state. Memory dependent regions are paths of code that define computation based on both register operands and memory state, with the requirement that the memory dependence be either completely or conservatively determined at compile time. The compiler first performs program-level alias analysis to identify such load instructions and annotates them as *determinable*, indicating that all potential store instructions can be determined at compile time. Both globally and locally-named structures are reused, whereas anonymous data structures are the subject of ongoing research. In our current implementation, neither stateless nor memory dependent regions are allowed to change the contents of memory.

4.2. Value Profiling

The nature of detecting a repeatable sequence requires some estimation of the run-time behavior of a program. The most direct method is value profiling [2]. Value profiling is an effective method of finding the value recurrence and potential reuse of instructions. Applying dynamic information with formal analysis [15] can also find relationships among computations and understand the fundamental source of program redundancy and predictability.

As the initial implementation of the proposed mechanism is completely directed at compile time, an accurate estimation of program reuse potential is essential. The Reuse Profiling System (RPS) was developed as a result of this work and is designed to report accurate reuse information for three components: instruction-level repetition, reusability for memory operations, and cyclic computation recurrence. Instruction-level reuse information consists of the frequency of individual values and the recurrence of values within a set time interval. Memory reuse information consists of the frequency of updates to the referenced memory locations of each memory instruction. Cyclic computation recurrence is gathered by profiling the input registers at the

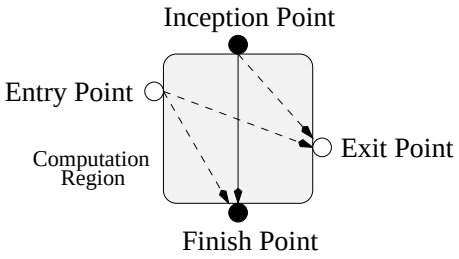


Figure 7. Abstract reusable region.

start of cyclic region invocation, by recording the number of iterations for each loop invocation, and by relating the individual memory reuse information of every cyclic iteration to the reusability of cyclic invocation.

4.3. Reusable Computation Regions (RCRs)

Several constraints exist in developing effective tasks for the reuse mechanism. The essence of our reuse scheme is that the compiler directs the hardware to regions of code that have reuse potential and can be conveyed through the instruction set architecture modification previously mentioned. Therefore, to use the mechanism effectively, the regions selected are those that have the maximal number of reusable paths (acyclic or cyclic) between a starting and ending point. The reusable computation of such paths is a subgraph of a deterministic computation region, called a *reusable computation region* (RCR). Figure 7 illustrates an abstraction of the reusable computation region concept. A reusable computation region delineates the section of program code that will be dynamically reused in the CCR framework and involves four region points:

Inception Point Starting point for memoization mode and location for reuse instruction.

Finish Point Ending point for memoization mode. Any computation within the path between the inception point and finish point can be reused.

Exit Point Side exit from computation region and termination of memoization mode. No reuse along paths from inception to exit point. Exit points do not necessarily exist for RCRs.

Entry Point Side entrance to reusable region that is not involved with reuse or memoization of computation. Entry points do not necessarily exist for RCRs.

4.4. Reuse Selection Policy

After generating reuse profiling information, the reuse characteristics of the code segments are known. Our current process of selecting reusable computation regions is heuristic-based and divides the computation into cyclic and acyclic region formation.

Cyclic region formation. Cyclic reusable regions are identified by detecting inner-nested loops with deterministic computation. This restricts the loops from altering memory state with store and subroutine instructions. Similarly, load instructions within the loop must be classified as determinable. These same regions are identified earlier by the profiling system in such a way that reuse information is gathered for each invocation of the loop. The cyclic profiling information is used to check that a loop has a greater than 40% opportunity to reuse results and that greater than 60% of the loop invocations have multiple loop iterations.

Acyclic region formation. The decision process for acyclic computation regions consists of five primary steps: seed selection, successor formation, predecessor formation, and subordinate path formation, and reiteration of the previous formation steps. The first step is to select a starting instruction for creating a computation region, known as the reuse seed. A seed instruction is selected from the set of all instructions within a function, ordered by the weight of the instruction execution, reuse potential, and the number of dependent instructions within instruction's basic block.

The second step is to extend the region from the reuse seed by selecting a path of reusable successor instructions. Selection of each successor is based on three criteria: instruction reusability, region inputs, and region accordance. An instruction is considered reusable if the weight of the top k recorded executions detected during profiling account for a large fraction of the instruction execution. Profiling support allows the ten most recent instruction executions to be maintained. Load instructions must satisfy two additional conditions: (1) the memory location referenced is reusable (defined by the frequency that stores access a location used by the load) and (2) be annotated as determinable. Control flow transitions between basic blocks are considered likely if the weight of the control flow edge is 60% of the weight of instruction i, $Exec(i)$. Otherwise, the invariance of reusing both branch operands is used to select the successor path. Essentially instruction-level profiling information is used to find the individual repeating instructions and to construct large regions of potential reuse in a bottom-up fashion. Therefore, an instruction i is reusable to the region if it first satisfies the heuristic functions shown below.

$$Reuse(i) = \left(\frac{Invariance[k](i)}{Exec(i)} \geq R\right) \quad (1)$$

$$MemReuse(i) = \left(\frac{Valid(i)}{Exec(i)} \geq R_m\right) \quad (2)$$

Empirical evaluation found that setting R and R_m to .65 and the number of invariant values to five produces good instances of reusable computation. Lower values tend to admit too many instructions in the region that are not successfully reused in reasonably sized CRBs.

The region input heuristic is used to determine if the instruction inputs overlap with the inputs of other instructions

already selected. Similarly, when considering a successor instruction, value analysis is performed to detect an occurrence when a source register value is confined to a limited set of values with the currently selected region. Finally, the total number of live-in and live-out registers within a computation region are limited to eight. The region accordance heuristic is used to prevent the inclusion of memory instructions to the region that increase the potential of invalidating the computation region already selected. As such, the accordance heuristic limits the number of distinguishable memory elements to four in order to reduce the creation of ineffectual memory dependence regions. Early experimental variations on the accordance heuristic revealed this setting to be productive.

These three characteristics find successor instructions that have good individual reuse, minimize the unnecessary invalidations, and minimize the number of input register dependences to the computation. The selection process attempts to reorder instructions to create larger reuse sequences. This prevents the original program ordering from hiding potential reuse. The process of adding successors to the region continues until the successor path can no longer be extended using the successor heuristics.

The third step of RCR formation is to expand the computation path by adding predecessor instructions that flow to the original reuse seed. The conditions for adding a predecessor instruction are analogous to the conditions of successors. The successor and predecessor points define the principle reuse path in the control flow graph representation. The fourth step is to add subordinate paths of reuse defined along the principle path. Such paths are selected by applying the similar heuristics as the main path selection algorithm. Side entrances, other than the inception instruction point to the selected set of paths are annotated as entry points. Exit points are defined by all branch instructions that are directed to code outside of the selected set of paths. The final step of the reuse selection policy is to continuely repeat the process of growing successors, predecessors, and subordinate paths until the region can no longer be expanded. By analyzing the newly formed region contents after each interval, the value-flow analysis heuristics are able to improve the reuse opportunities. At the end of iterative process, the top-most resulting instruction is established as the inception point for the reusable computation. The bottom-most instruction is defined as the finish point.

The overall algorithm consists of cylic region formation, acyclic region formation, and region transformations. The transformations remove subsumed regions, partially duplicate benefical regions, and combine regions using minimal tail duplication. A complete description of the reuse heuristics and selection algorithm is available in [3].

5. Experimental Evaluation

5.1. Methodology

The IMPACT compiler and emulation-driven simulator were enhanced to support a model of the proposed architecture framework and the region formation techniques respectively introduced in Section 3 and Section 4. The benchmarks used in all experiments consist of SPECINT92, SPECINT95, UNIX, and MediaBench programs. The base level of code consists of the the best code generated by the IMPACT compiler, employing function inlining, superblock formation, and loop unrolling.

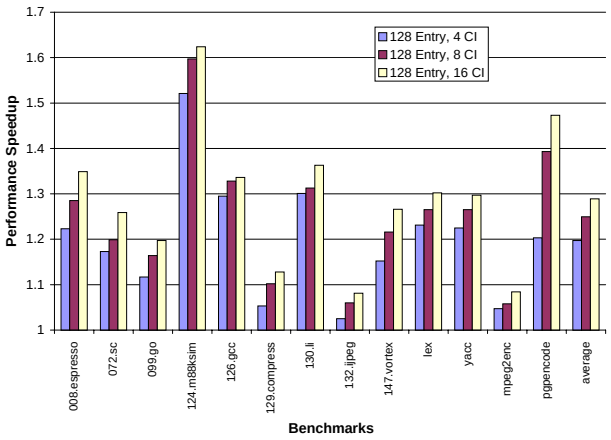
The base processor modeled can issue in-order six operations up to the limit of the available functional units: four integer ALU's, two memory ports, two floating point ALU's, and one branch unit. The instruction latencies used match the HP PA-7100 microprocessor (integer operations have 1-cycle latency, 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, and a miss penalty of 12 cycles; 4K entry BTB with 2-bit saturating counters, and a branch misprediction penalty of eight cycles. Failure to correctly reuse computations causes the processor to experience a delay similar to the branch misprediction penalty. For our simulations, the computation buffer had 32, 64, or 128 direct-mapped entries with 4, 8, or 16 computation instances (CIs) per entry. Each CI supports an input and output 8-entry register array.

5.2. Results and Analysis

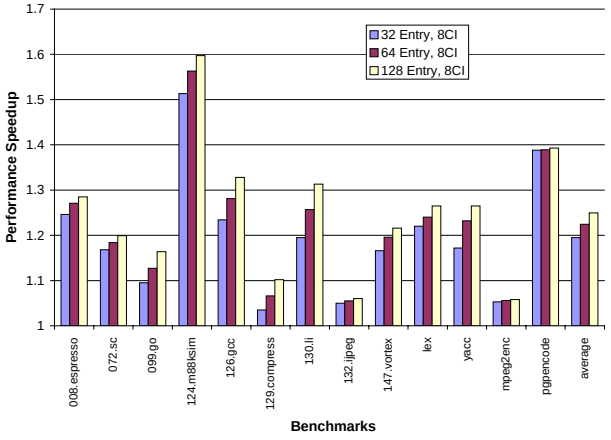
Three categories of results are presented. The overall performance of the CCR approach is first examined. Second, some of the characteristics of the reusable computations are presented. Third, some general results relevant to the compiler-directed scheme are evaluated.

Performance. The overall cycle-time speedups for the compiler-directed approach are presented in Figure 8. Two variations in the CRB design are evaluated: variation in the number of computation instances per computation entry and variation in the number of computation entries. Performance is reported as speedup which is derived by dividing the execution cycles for the base architecture by that of the architecture with the CCR framework.

The first CRB variation considered is the number of computation instances per computation entry. Figure 8(a) presents the effect on performance for varying the number computation instances. On average, a processor with 128 computation entries has speedups of 20% for a computation entries with 4 CIs, 25% for 8 CIs, and 30% for 16 CIs. The reuse of computation using the CCR approach is most effective for *124.m88ksim*, where there are a number



(a)



(b)

Figure 8. Speedup for processor with CCR support (a) varying the number of instances and (b) varying the number of entries.

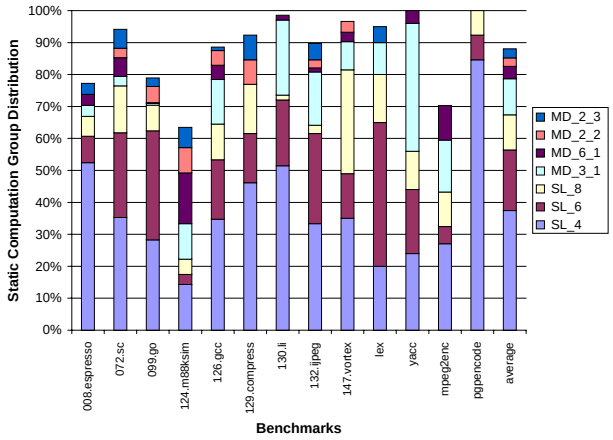
of substantial computations that are frequently reused. Variation in the number of computation instances substantially increased the performance speedup of *pgpencode*. This was mainly due to the type of computations being reused. In this benchmark, a number of stateless computation regions were formed using the heuristics based on average reuse occurrence, but the computations have considerable dynamic variation. A large number of computation instances is able to effectively handle this variation. Overall, the average performance improvements indicate that the 128-entry computation buffer with 8 CIs per entry is potentially the most cost effective. In improving the processor performance, the CCR approach eliminates an average 40% of the dynamic instruction repetitions that occur on the base processor configuration.

The next CRB variation considered is the number of computation entries. Figure 8(b) illustrates the effect on performance for varying the number of computation entries. For a model with 8 computation instances, the average speedups are 20% for 32 computation entries, 23% for 64 computation entries, and 25% for 128 computation entries. The benefits of reuse are sustained for even a small number of computation entries. On average, the majority of benchmarks are characterized by a small number of reusable computations that account for a large portion of the overall execution time. This indicates that the amount of region-level reuse can potentially be exploited with a moderate number of computation entries.

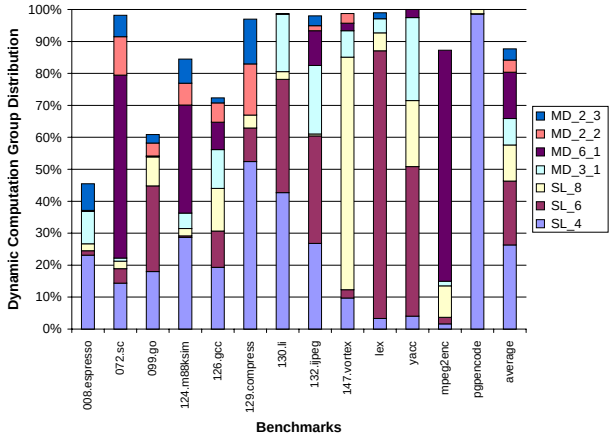
Reuseable Computation. Several important aspects of the computations being reused in the CCR approach were investigated. The first is the class of computation and the second is the type of computation. The classification of computation is either Stateless (SL) or Memory Dependent (MD). The classes can be subdivided into groups indicating the general input type of the computation. The group naming convention of SL_{*num_input*} is used to indicate the group of stateless computations dependent on *num_input* registers. Similarly, MD_{*num_input*}_{*num_mem*} is used to indicate the group of memory dependent computations dependent on *num_input* registers and *num_mem* distinguishable memory structures. The number of distinguishable memory structures for each MD group is determined by the compiler's region formation heuristics. Overlapping group results are included within some of the group entries. For instance, group SL_8 includes the computations of SL_7 but not SL_6 since the group SL_6 is also presented. Similarly, MD_6_1 includes all computations dependent on a single memory structure and up to six register inputs.

Figure 9 compares the static and dynamic distribution of seven groups of computation that account for the most reuse of program execution. Figure 9(a) presents the static distribution of the computation groups. On average, nearly 90% of the computations are included within the seven selected groups. The distribution indicates that stateless computations account for an average of 65% of the static computations created by applying the current RCR heuristics to optimized programs. Evaluation determined that the acyclic formations of the the seven computation groups replace the execution of an average of 10 instructions. Figure 9(b) illustrates the dynamic distribution of the seven computation groups. On average, the dynamic execution of stateless computation regions accounts for 60% of the reuse execution. Several of the benchmarks are able to effectively reuse computation results stored in memory by employing the computation groups MD_3_1 and MD_6_1. These results indicate that the CRB could be designed to have only a portion of the computation entries with memory reuse capabilities.

Figure 10 presents the amount of reuse execution distributed by the percentage of active computations. Four sets are used to indicate the amount of dynamic reuse generated



(a)



(b)

Figure 9. Computation group (a) static distribution and (b) dynamic distribution.

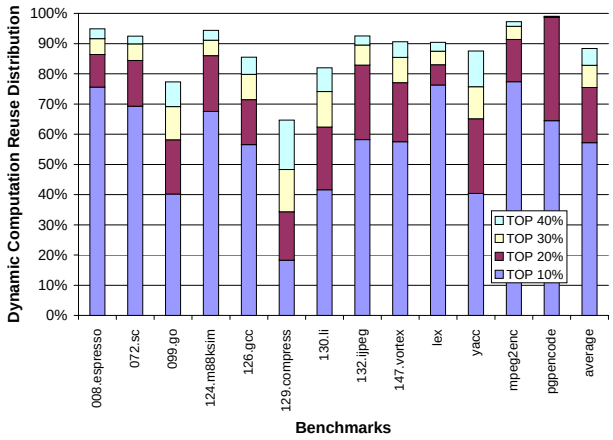


Figure 10. Dynamic reuse distribution.

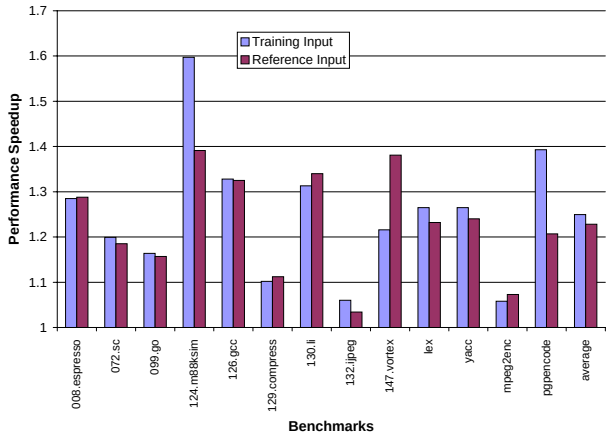


Figure 11. Performance for training and reference input data sets.

by 40% of static computations. Each set includes 10% of the computations which are sorted by their contribution to the total reuse execution. For instance, TOP 10% indicates the reuse attributed to the top 10% of contributing computations. We see that the cumulative results indicate that 40% of static computations account for nearly 90% of total reuse. *129.compress* is one of the few benchmarks for which different reuse distributions exist. In this case, each of the program computations are closely weighted in the amount of reuse execution that they contribute. Nevertheless, the average characteristics of Figure 10 is a good indicator of the possibility of exploiting redundancy with limited hardware.

General Approach Evaluation. Due to the nature of determining reusable computation at compile time, the performance potential of the CCR framework depends on utilizing effective heuristics and having accurate run-time estimation of program behavior. To determine the significance of using value profiling information outlined by our approach to make these decisions, we measured the performance of an input data set different from the one used to determine reusable computation. The training and reference input sets were then evaluated, and the results compared to each input's respective base performance. Figure 11 examines these results for a 128-entry CRB model with 8 CIs per entry. The average performance speedup for the training input set is 26%, and the average for the reference input set is 23%. Although on average there is a reduced relative speedup for the reference input set, several of the benchmarks achieved higher relative speedups for the reference input set. In addition, the average amount of instruction-level redundancy eliminated using the reuse architecture averaged 33%. This level is close to the average 40% instruction repetition eliminated during the evaluation of the training input set, and helps illustrate the general applicability of directing the reuse of computation at compile time.

6 Summary

In this paper, we introduced a compiler-directed approach for exploiting dynamic redundancy with several advantages. The approach uses the compiler to decide and annotate code segments with the potential for reuse. This enables the underlying hardware architecture to capture the reuse potential in regions of code rather than basic blocks and individual instructions, thereby exploiting more substantial opportunities. Additionally, the approach captures the redundancy of high-level computations that do not possess instruction-level repetition. We evaluated the effectiveness of the proposed approach using several different resource models, varying the size of the computation reuse buffer and the number of computation instances recorded for each computation. The resulting speedup of the approach measured with a moderate architecture model (128-entry CRB with 16 computation instances per entry) achieved an average 30% speedup.

This paper presents only an initial study of the ability of the compiler-directed approach to exploit dynamic redundancy. There is a considerable amount of future work to investigate in both the compiler and architecture domains. In the compiler domain, the aspect of directing the CCR architecture at the function level could potentially reduce a significant amount of time spent executing calling convention and spill codes. In the architecture domain, we will investigate reuse buffers with nonuniform capacities and the use of value speculation techniques to hide the latency of validating reuse opportunities.

Acknowledgments

The authors would like to thank Scott Mahlke, Ben-Chung Cheng, John Gyllenhaal, Dan Lavery, Allan Knies, David August, and all the members of the IMPACT compiler team for their valuable comments. This research has been supported by the National Science Foundation (NSF) under grants CCR-96-29948 and CCR-98-31551, and by the Intel Corporation under an unrestricted gift. Dan Connors was also supported by an Intel Fellowship during the academic year of 1998-1999.

References

[1] R. Bodik and R. Gupta. Complete removal of redundant computation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 1–14, June 1998.

[2] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 259–269, December 1997.

[3] D. A. Connors and W. W. Hwu. A compiler-directed computation reuse architectural framework. Technical Report IMPACT-99-04, IMPACT, University of Illinois, Urbana, IL, September 1999.

[4] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[5] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte. Value speculation scheduling for high performance processors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1998.

[6] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 270–280, December 1997.

[7] D. M. Gallagher, W. Y. Chen, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.

[8] A. Gonzalez, J. Tubella, and C. Molina. Trace-level reuse. Technical Report 47, Departament d' Arquitectura de Computadors, Universitat Politecnixa de Catalunya, July 1998.

[9] S. P. Harbison. An architectural alternative to optimizing compilers. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 57–65, March 1982.

[10] J. Huang and D. J. Lilja. Exploiting basic block value locality with block reuse. In *The 5thInternational Symposium on High-Performance Computer Architecture*, January 1999.

[11] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 226–237, December 1996.

[12] M. H. Lipasti and J. P. Shen. Value locality and load value prediction. In *Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, September 1996.

[13] S. E. Richardson. Caching function results: Faster arthmetic by avoiding unnecessary computation. Technical Report 92-1, Sun Microsystems Laboratories, 2550 Garcia Avenue, Mountain VIew, CA 94304, September 1992.

[14] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[15] Y. Sazeides and J. E. Smith. Modeling program predictability. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, June 1998.

[16] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 194–205, June 1998.

[17] A. Sodani and G. S. Sohi. An empirical analysis of instruction repetition. In *Proceedings of 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–45, October 1998.

[18] A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 205–215, December 1998.