Hardware Support for Dynamic Activation of Compiler-Directed Computation Reuse

Daniel A. Connors[†], Hillery C. Hunter[‡], Ben-Chung Cheng*, Wen-mei W. Hwu[‡]

[†]Department of Electrical and Computer Engineering & Computer Science
University of Colorado
dconnors@colorado.edu

[‡]Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
hhunter,hwu@crhc.uiuc.edu

*Transmeta Corporation
Santa Clara, CA
bccheng@transmeta.com

ABSTRACT

Compiler-directed Computation Reuse (CCR) enhances program execution speed and efficiency by eliminating dynamic computation redundancy. In this approach, the compiler designates large program regions for potential reuse. During run time, the execution results of these reusable regions are recorded into hardware buffers for future reuse. Previous work shows that CCR can result in significant performance enhancements in general applications. A major limitation of the work is that the compiler relies on value profiling to identify reusable regions, making it difficult to deploy the scheme in many software production environments. This paper presents a new hardware model that alleviates the need for value profiling at compile time. The compiler is allowed to designate reusable regions that may prove to be inappropriate. The hardware mechanism monitors the dynamic behavior of compiler-designated regions and selectively activates the profitable ones at run time. Experimental results show that the proposed design makes more effective utilization of hardware buffer resources, achieves rapid employment of computation regions, and improves reuse accuracy, all of which promote more flexible compiler methods of identifying reusable computation regions.

1. INTRODUCTION

In the Compiler-directed Computation Reuse (CCR) approach [7], the compiler identifies code regions whose computation can be reused during execution to eliminate dynamic redundancy [12, 15, 16]. The instruction set architecture provides an interface for the compiler to communicate the scope of each region to the hardware. During run time, the microarchitecture of the approach records the execution results of the reuse regions. Results show that the approach can eliminate a large number of dynamic instructions, resulting in much higher execution speed and efficiency.

The compiler-directed approach uses a hardware caching structure called the Computation Reuse Buffer (CRB) that interacts with CCR instruction extensions to achieve reuse at run time. The CRB structure consists of *computation entries* that support the reuse for a particular region by maintaining an array of dynamic computation information for different execution instances of the region. Each *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 region is reusable when a computation instance within the region's designated computation entry matches the input register values with a previous recorded execution of the region and the input memory state has not been invalidated.

In the base CCR approach, code segments called Reusable Computation Regions (RCRs) are selected at compile time using profile information that estimates the expected amount of reuse that will occur during execution. Because the hardware structure of the base CCR approach always attempts to reuse previous computation results for all annotated regions, the regions contend for the same computation reuse resources regardless of each region's importance to program execution. Thus, the compiler must select only those regions whose reuse at run time will result in the most benefit.

Figure 1 shows the performance achieved for a six-issue processor using two CRB models. The first model is a 32-entry CRB with 16 computation instances per entry, and the second model is a CRB with infinite resources that maintains all execution results for every region selected at compile time. Since the infinitely resourced CRB does not have resource contention, the compiler designates more computation regions by lowering the reuse behavior requirements. The performance results are relative to a base processor without reuse support. On average, the 32-entry CRB design captures 40% of the potential speedup of the infinite CRB. The performance difference occurs because the compiler faces an undesirable tradeoff between missing reuse opportunities and exhausting CRB resources. The potential performance benefit achieved by eliminating resource contention provides motivation to enhance the CCR approach to allow the compiler to perform more aggressive region identification and to enable dynamic computation reuse activation.

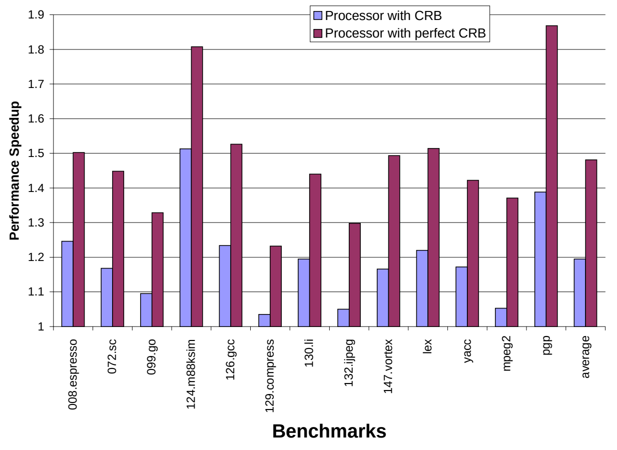

Figure 1: Performance for processor with base CRB and a CRB with infinite capacity.

Although the base CCR approach can eliminate many significant instances of redundant execution, there are additional opportunities that require run-time examination of region behavior to completely eliminate dynamic computation redundancy. For instance, virtually all programs go through a series of execution stages characterized by changes in the properties for code, the data, or both. Similarly, programs such as compilers, interpreters, and graphics engines exhibit phase behavior, having different modes of operation for different inputs [11]. In analyzing run-time computation reuse, experimental observations indicate that reuse behavior often occurs in distinct phases. The use of profile information by compiler-directed mechanisms can hide reuse opportunities because profile-guided decisions may not be representative of all workloads. More importantly, the use of profile information in many systems is not feasible due to constraints in software development. For these reasons, it is imperative that a system adapt to variations in program behavior.

Periods of region executions with successful reuse are called *reuse intervals*. An enhanced dynamically-activated CCR scheme can take advantage of reuse intervals by favoring the utilization of the computation buffer resources for those regions. Ideally, only computation regions demonstrating reuse success would be deployed at run time. Reuse interval behavior can be seen for *008.espresso* running the bca training input from the SPEC92 benchmark suite. Figure 2 shows the reuse execution for three computation regions collected over two million executions. Each point indicates whether the region had a computation miss (no cached computation results), a computation hit (valid computation result from a previous execution), or no attempt (the region was not executed). All three regions have initial periods with large numbers of reuse misses. During this time, computation instances are being stored for future execution.

The behavior of the first region indicates two medium-sized reuse hit intervals with intervening periods of reuse misses. The second and third regions have longer reuse intervals, which occur after the initial cold start period. However, the third region experiences a long period of time without any execution. The identification of reuse intervals can be made over different lengths of program execution time. Hardware-based activation techniques can most accurately detect and deploy regions based on reuse intervals where the time steps are measured in region executions rather than instruction executions. The data in Figure 2 motivates dynamic management of computation reuse because they show that regions execute with periods of reuse and multiple regions compete for the same computation reuse resources.

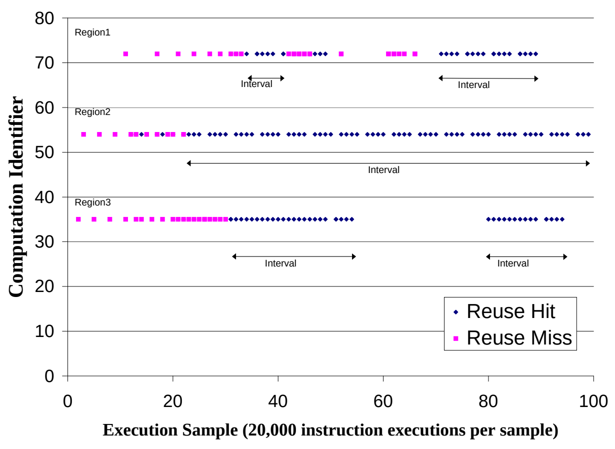

Figure 2: Computation region behavior for *008.espresso*. Data points indicate either a computation reuse miss, reuse hit, or no attempt.

The third motivation for enhancing the CCR approach is the general observation that the number of computation instances that a region requires over an interval of execution varies with the region type. Regions exhibiting *regular variation* typically require a constant number of computation instances over the lifetime of the program. Such regions can be assigned to computation entries with a specific computation capacity, thereby reducing the hardware costs of providing a large number of instances per CRB entry. Conversely, experimental results indicate that 70% of regions have *irregular variation* behaviors that warrant different computation instance capacities at different times. To effectively utilize computation resources, the CRB needs to adapt the allocation of computation instances to run-time requirements.

The final motivation for CCR improvement is that the base approach relies on profile-guided heuristics to identify regions that are potentially reusable during the execution of the program. Specifically, value profiling techniques [3] determine the regions that instruct the hardware to effectively reuse the computation defined within these regions. Since the reliance on value profiling can hamper the use of the CCR approach in software production environments, it is desirable to eliminate the need for profiling at compile time.

1.1 Overview

Several proposed architecture techniques can realize significant performance benefits by adapting to run-time behavior. Branch prediction and cache management have been the primary areas where run-time information has been applied. However, trends in dynamic optimization [1] and run-time hotspot detection [13] indicate other exciting avenues to exploit run-time behavior. In these cases, run-time information allows for a more effective use of processor resources. Likewise, giving the compiler-directed computation reuse approach access to run-time information would allow it to adapt to program trends. First, the reuse of a computation region can be activated at run time rather than at compile time. Second, the run-time information can be consulted to effectively assign resources to those regions generating substantial performance improvements. Third, the run-time variation statistics would allow effective allocation of reuse buffer resources in the presence of hardware with varying recording capacities. Finally, dynamic management techniques allow the compiler to introduce computation regions that are not guided by profile information.

The limitations of the original CCR approach are addressed by a new hardware-based system that activates reusable computations within program execution at run time. The system uses three integrated structures. The first hardware structure, the *Reuse Sentry*, detects regions with significant execution frequency and controls the deployment of active computation regions. The *Evaluation Buffer* evaluates candidate computation regions for potential reuse. This structure monitors selected regions to determine if the regions should be placed in the reuse buffer. Finally, the third component, the *Reuse Monitoring System*, examines the behavior of computation regions assigned to the computation buffer and makes adjustments to its allocation. The system introduces architectural support for eliminating the inherent dynamic redundancy occurring in programs due to aspects of programming languages and application workloads.

2. DYNAMIC MANAGEMENT SYSTEM

The proposed hardware support for dynamic computation reuse management uses three stages to activate a computation region. The stages perform detection, evaluation, and examination of beneficial computation reuse. First, the region must meet a minimal execution requirement to warrant consideration for CRB resources. Second, the region must be evaluated to determine the likely benefit of the reuse mechanism. Finally, the variation of the computation region is examined to determine the most effective way to allocate the CRB resources. These stages act as a run-time confidence mechanism to accurately select regions for the computation buffer resources, and result in improved CRB utilization and reuse accuracy. The stages are collectively constructed in the Dynamic Computation Management System (DCMS) which consists of three hardware components:

Reuse Sentry (RS) collects execution counts for computation regions and identifies candidate regions. The sentry uses a Candidate Execution Counter to determine the activity of observed regions and a Reuse Deployment Buffer to deploy region execution.

Evaluation Buffer (EB) evaluates candidate computation regions for potential reuse. The structure monitors candidate regions by recording execution behavior in specialized hardware buffers.

Reuse Monitoring System (RMS) examines the behavior of regions in the computation buffer and assesses allocation of CRB resources. The system removes computation regions from the CRB and directs alternate computation regions to utilize the CRB resources.

Figure 3(a) shows the base CRB model and Figure 3(b) illustrates the proposed dynamic computation management system. In Figure 3(a), computation regions always attempt reuse and subsequently contend for entries in the CRB. In the enhanced design of Figure 3(b), the CRB resources are selectively assigned to regions with persistent reuse.

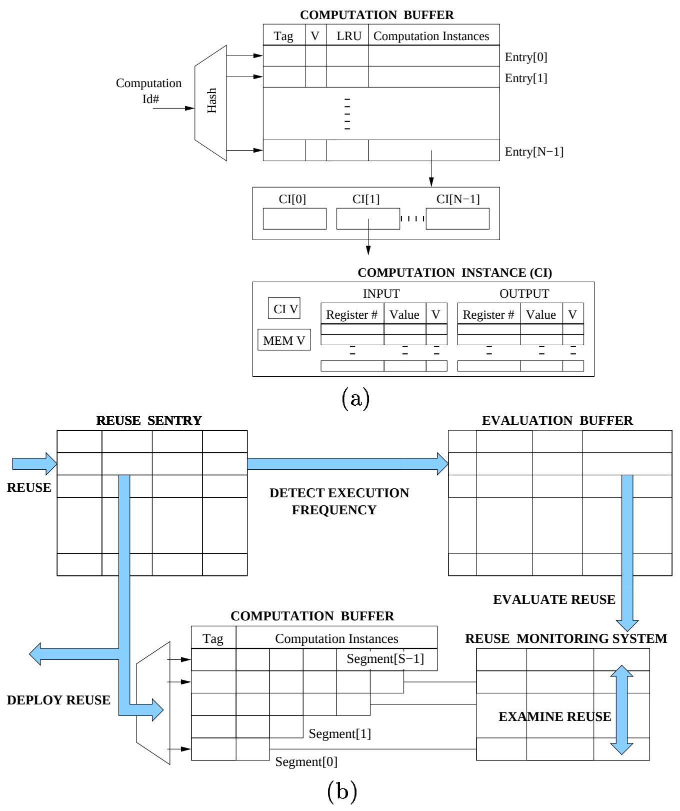

Figure 3: The Computation Reuse Buffer (a) and the Dynamic Management System (b).

The Reuse Sentry and Evaluation Buffer can be located off the critical path of the processor pipeline because their resources do not directly affect the use of previously cached results. However, the Reuse Monitoring System and the operation of the CRB require close interaction with the processor datapath. These systems cannot tolerate a large access or update latency because the access latency of the computation entries and their respective computation instances is inversely proportional to the performance benefit of the CCR approach [7]. The following sections describe the components of the dynamic management system.

2.1 Reuse Sentry

The first step in effective management of the computation reuse resources is detection of frequently executed computation regions. Such regions can be easily identified in hardware by detecting a high execution frequency over a particular time interval. By examining all region executions over the same interval, computation resources can be accurately allocated to regions with different requirements.

The *Reuse Sentry* (RS) structure collects the execution history and efficiently deploys reuse. This hardware is named

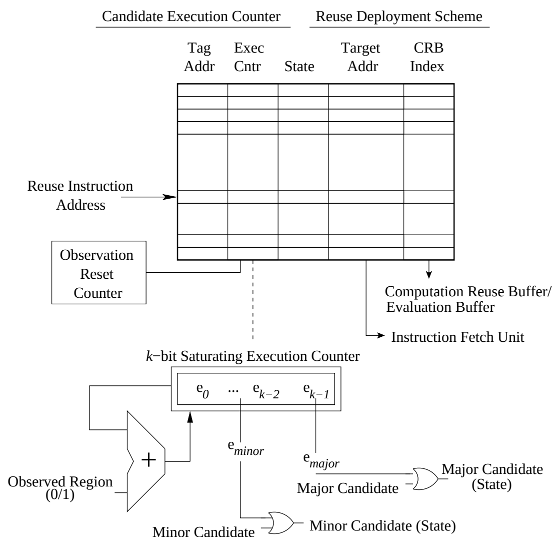

Figure 4: Reuse Sentry hardware.

for its function as a *sentry*, or guard, that prevents the passage of unauthorized regions. As depicted in Figure 4, the RS structure is indexed by the reuse instruction address and contains several fields: address tag, execution count, state information, predicted address, and computation entry index. The RS structure also includes a global *Observation Reset Counter* used to evaluate all region executions. The activities of the RS structure are divided between the Candidate Execution Counter and the Reuse Deployment Buffer.

2.1.1 Candidate Execution Counter

The Candidate Execution Counter (CEC) primarily detects regions with frequent execution. Regions that execute often are given higher priority for resources than regions with low frequencies. A secondary function of the CEC is to record the position of each region in the DCMS. The position follows the transitions of Figure 5, which shows four states: *observed*, *candidate*, *active*, and *inactive*.

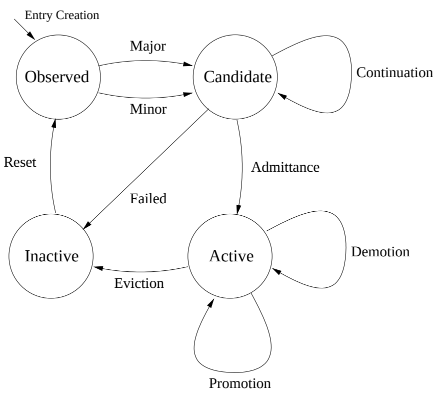

Figure 5: Region states and transitions.

When the processor executes a reuse instruction, an entry is created in the RS, and the instruction address is designated as the index. The initial creation of an entry classifies the region as an *observed region*. In this state, the region is monitored to determine if it frequently executes. Regions that execute frequently over a short time interval are excellent candidates for reuse resources. If executed frequently during the observation state, a region may be passed to the Evaluation Buffer, and is then referred to as a *candidate region*. In order to detect candidate regions, the execution counter of a region's RS entry is incremented on each execution of the region. The execution counter can exceed two predefined levels, the *major* and *minor candidate thresholds*, each of which is associated with a bit in the execution counter. When the counter bit corresponding to a threshold is set for the first time, a candidate flag is set in the state field for the duration of the observation time. The major threshold indicates regions with a dominant number of executions, while the minor threshold indicates a lower execution frequency. The distinction between major and minor thresholds aids determination of the best candidate regions observed in a particular interval.

To ensure that only frequently executing regions are marked as candidates, the RS is periodically flushed. The Observation Reset Counter is used to establish a time interval, called the *observation interval*, for periodically refreshing the entries of the RS that have not surpassed any candidate threshold or been classified as *inactive*. The Observation Reset Counter is incremented each time an inactive or observed region is executed. The inactive classification is assigned by the evaluation and examination stages of the DCMS, and is used to designate regions that previously occupied the CRB, but had poor reuse behavior.

When the major candidate threshold is surpassed, hardware logic attempts to create a region entry in the Evaluation Buffer (EB) and change the region state from observed to candidate. If the EB does not have sufficient entries to handle the request, another attempt is made at the end of the observation interval. In this case, the RS is scanned for all entries meeting the major candidate threshold. This means that relative to the observation interval, entries in the evaluation buffer are made both asynchronously and synchronously. If the EB has capacity remaining after obtaining major candidate regions at the end of the interval, then the RS is searched for entries meeting the minor threshold.

2.1.2 Reuse Deployment Buffer

The second function of the Reuse Sentry is to provide efficient deployment of compiler-directed computation reuse. The basis of the CCR approach is placement of a reuse instruction at the entrance to a large region of code that exhibits computation redundancy. The reuse instruction is formulated as a branch instruction with two potential locations for the next instruction to be executed: the fall thru or the taken location. The control resolution is based on whether valid computation results are stored for the region.

In the dynamically-managed approach, regions classified as *inactive*, *observed*, or *candidate*, have not been assigned computation reuse resources and execute using processor resources. Otherwise, a region is *active* with CRB resources assigned to it and there is a good chance that region execution can be bypassed by reusing previous computation results. In order to improve deployment of computation reuse, the Reuse Deployment Buffer (RDB) is constructed similar to a *Branch Target Buffer* (BTB) and predicts reuse outcome. For active regions, the target address field in the

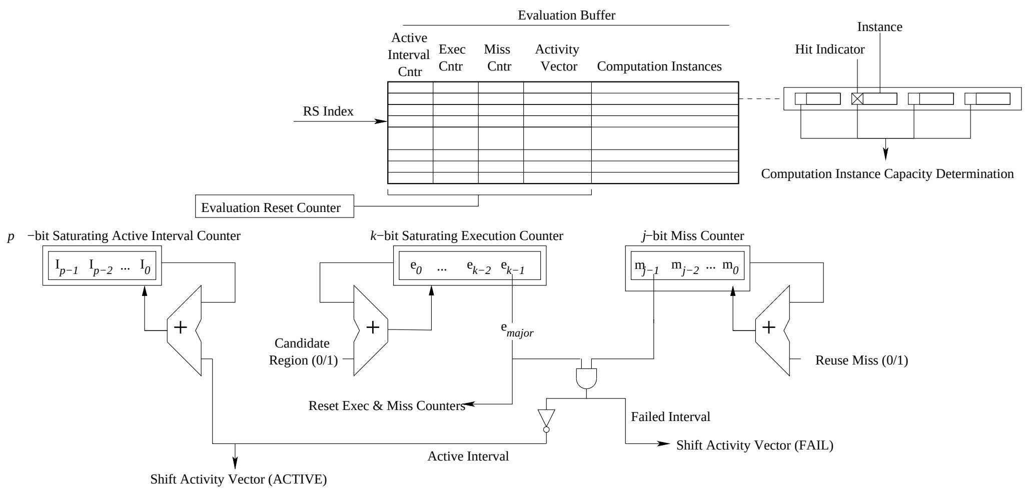

Figure 6: Evaluation Buffer hardware.

RS entry of a region is used to predict the target of the reuse instruction as the next instruction following the entire computation region. Otherwise, the RS predicts not-taken, and the execution continues with the next sequential instruction. The RS prediction reduces the delay in determining whether the computation region has been assigned resources.

2.2 Evaluation Buffer

After the Reuse Sentry identifies candidate regions, the regions are evaluated to determine their reuse behavior and whether they warrant computation reuse resources. A candidate region must satisfy two criteria to be transitioned from the EB (candidate state) into the CRB (active state). First, the candidate must exhibit a high percentage of reuses, called an *active reuse interval*, occurring over a minimum execution time, called the *active evaluation interval*. The minimum percentage of successful reuse over the time interval is called the *active reuse threshold*, while the actual reuse percentage over the interval is the *active reuse percentage*. The number of failed reuse attempts is tracked to determine whether an active interval exists. Second, the candidate region must have a minimum number of active intervals during the *evaluation interval*, a number of all candidate region executions tracked by incrementing the Evaluation Reset Counter for each candidate execution. The evaluation interval resets the buffer entries, allowing new regions to be evaluated. The number of active intervals exhibited by a region during the evaluation interval is called its *active interval count*.

The Evaluation Buffer structure is shown in Figure 6 with entries that contain the following fields: active interval count, execution count, miss count, activity vector, and a computation instance array. The active interval count maintains the number of active intervals experienced during the evaluation interval and is used in assigning activation priority. The execution counter tracks the number of executions in the currently evaluated active reuse interval. Miss count represents the number of reuse misses for the current active interval. The execution counter is implemented as a roll-over counter that is initialized to the minimum value. The counter increments for each execution candidate region execution and an active interval is counted if the number of failed opportunities for reuse does not exceed the *failed threshold*. The result of evaluating an active interval (active/failed) during every active evaluation interval is used to maintain an *active difference* in a special shift register called the *activity vector*, shown in Figure 7. This register is shifted in the active direction for every active interval achieved and shifted in the opposite direction for failed intervals. The vector maintains the difference in the number of active and failed intervals and supports the evaluation of EB entries.

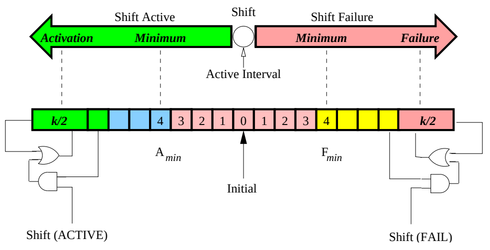

Figure 7: Activity vector.

At the end of the evaluation interval, EB entries are removed under two conditions. First, the entry is removed if there is not an active interval count and a positive *active difference* in the activity vector. This allows regions with only marginally reuse to removed. Upon reaching the *maximum activation threshold*, attempts are made by the EB system to place the respective regions in the CRB. Similarly, at the end of the evaluation interval, region entries meeting a *minimum activation threshold* are compared with entries in the CRB for opportunities to revise CRB resource allocation. If no favorable opportunities exist, the EB entry is reset, but the region remains under evaluation (*continuation*). Second, the entry is immediately removed if the activity vector reaches the *maximum failed threshold*. Entries with an activity vector indicating a *minimum failed threshold* are possibly replaced depending on the requests of the RS. At the end of the evaluation interval, entries with activity vectors indicating activity below the minimum active and minimum failed thresholds are also removed. Any of the above entry failures results in the region state being changed to *inactive*, while admittance to the CRB results in *active* state assignment.

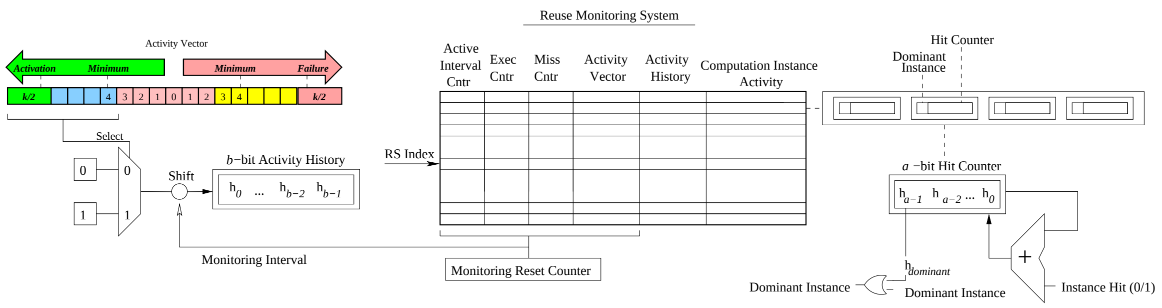

Figure 8: Reuse Monitoring System hardware.

The computation instance array of the EB entry differs in two ways from the traditional computation instance array. First, the input set only consists of input register operands and their respective values, and the validation of memory state used by the computation. In order to minimize the dynamic management support hardware, the set of output register operands and their respective result values are not stored. Second, the array contains a single bit, called an *instance indicator*, for each computation instance activated during the evaluation process. The instance indicators serve to make an assessment of the number of computation instances necessary for the region. The EB migrates computation regions from the candidate to active state based upon availability of computation reuse buffer resources. The best candidate region entry is the one with the largest difference between its number of active and failed intervals. The number of set instance indicators is used to make the initial assignment to computation reuse entries. The computation reuse resources have a variable number of computation instances per computation entry, and the RMS informs the EB of available entries with their respective capacities. The EB decision is based first on matching the available capacity with the potential required capacity of entries and then on reuse evaluation which is decided by activation difference).

2.3 Reuse Monitoring System

The Reuse Monitoring System (RMS) determines if the computation resources designated for each active region are appropriate and revises the assignment of resources based on the observed run-time requirements. The computation reuse entries are divided into segments, each containing a different computation instance array capacity. Generally, the segmentation includes entries with differing numbers of computation instances per computation entry and allows the RMS to match the hardware capacity with a region's requirements (adapting to irregular variation). The RMS performs four entry revisions: *demotion*, *promotion*, *eviction*, and *admittance*. The process of *demotion* transplants a region to an entry with lower computation instance capacity, while *promotion* gives a region higher computation instance capacity. The RMS is responsible for *evicting* regions from the computation reuse buffer if they do not result in successful reuse. Finally, a region be *admitted* from the EB if its reuse behavior is more favorable than existing CRB entries.

Figure 8 illustrates the proposed Reuse Monitoring System. Each computation entry in the CRB has a respective entry in the RMS governs the replacement policy of the entry and the replacement policy of the computation instances. The policies are able to share the same hardware components within the entries since both operations are related to the run-time utilization and accuracy of the computation instances. To support appropriate revisions, reuse behavior information is collected in the same manner as the EB entries. Figure 8 illustrates the additional fields for collecting activity information: activity history and computation instance activity. A *monitoring interval* is computed using the *RMS Counter*, which is incremented for every active region execution. The monitoring interval periodically revises the CRB entries and imports new region entries from the EB.

Computation Entry Revision. For the RMS to determine region requirements, the hardware must be aware of the reuse accuracy of the computation placed in each computation entry. In addition, each entry maintains an *activity history* field that contains an b-bit shift register for recording the history of achieving the minimum activation percentage during each of the last b monitoring intervals. Bit position $history_0$ represents the current monitoring interval and position $history_{b-1}$ represents the activity level from $b-1$ monitoring intervals ago. In addition, the CIA counter information is used to estimate the needs of the computation region during revision of resource assignment.

Computation Instance Replacement. The RMS hardware provides reuse accuracy by tracking the activity of the computation instances within each computation entry. A traditional replacement policy for the instances is least recently used (LRU), however, some instances, called *Dominant Instances*, of computation occur frequently over the lifetime of a region. The accuracy of region reuse can be greatly enhanced by allowing the dominant instances to remain in the CRB. To do this, *Computation Instance Activity* (CIA) saturating counters are incremented for each successful reuse of an instance. When the counter saturates, the *Dominant* field is set and remains set for the entire monitoring interval. On a computation instance miss, non-dominant regions are selected using a LRU policy.

Revisions are performed at the end of the monitoring interval and upon detection of an activity vector with excessive failures. The proposed implementation steps (followed in order of appearance) are summarized in Table 1. The timing column refers to revisions taking place on the reset of either the evaluation or monitoring intervals. The *segment_level* (SL) refers to the partitioned segment of the CRB, a higher

Revision	Timing	Condition
Eviction	Asynchronous	MaxFailure Vector && !Segment_Entry_Available[SL+1→MAX]
	Synchronous	MinFailure Vector && DC && DC Vector > RMS Vector
Demotion	Synchronous	MinActive Vector && Dominant Instances < NumInstances[SL]/2
Promotion	Asynchronous	MaxFailure Vector && Segment_Entry_Available[SL+1→MAX]
	Synchronous	MinFailure Vector && Segment_Entry_Available[SL+1→MAX]
Admittance	Synchronous	DC MaxActive Vector && Segment_Entry_Available[Dominate Instance Level→MAX]
	Asynchronous	DC MinActive Vector && Segment_Entry_Available[Dominate Instance Level→MAX]
	Synchronous	DC MaxActive Vector && DC Vector > RMS Vector && Population(RMS History) < 1/2 History Size && Population (recent RMS History) < 1/2 Population (RMS History)

Table 1: RMS region revision conditions.

segment level indicate segments with greater computation instances capacity. A *Desirable Candidate* (DC) refers to a region in the EB with a promising activation vector. When admitting regions from the evaluation buffer, the instance indicators are used to determine the base segment of the CRB to detect RMS entries. Admittance of a candidate region may preempt a region with a poor activity history.

3. COMPILER SUPPORT

To take full advantage of the dynamically-managed CRB hardware, a compiler should select all computation regions that could compute recurrent values during execution. This requires the identification of the different reuse behaviors that can occur for different input sets and modes of operation. In this section, profile-guided and static compiler algorithms that facilitate traditional software development and the CCR dynamic management system are presented.

3.1 Profile-guided Region Formation

By profiling an application on a set of sample inputs, representative run-time information can be conveyed to the compiler. This enables an optimizing compiler to increase application performance by transforming its code to achieve better execution efficiency for those sections of the program with the highest execution frequency. Optimization of applications based on run-time value invariance [1, 3] offer great potential in exploiting run-time behavior. Other dynamic techniques have focused on discovering invariant relationships between variables from execution traces [8]. Invariant value profiling was also used to prove the effectiveness of compiler-directed computation reuse [7]. In the original computation reuse approach, it was important to select only statistically beneficial regions because the reuse mechanism could not be selectively used at run time. However, when using the DCMS which can selectively enable computation regions, it is imperative that all regions with potential reuse behavior be annotated. Since many programs have different modes of operation for different inputs, collecting profiling information on a wide variety of inputs is an essential part of evaluating the effectiveness of the DCMS design.

To establish the effectiveness of the proposed DCMS scheme, value-invariance profiles were collected for benchmarks from *SPECINT92*, *SPECINT95*, UNIX utilities, and media applications [10] using a training input and a reference input. Region formation steps were applied to programs annotated with value-invariance information from the separate inputs. Table 2 shows the resulting region statistics. Columns 2 and 3 indicate the number of regions formed based on the training and reference inputs respectively. Using the number of regions formed based solely upon the training input set as a base, three fractions are calculated. The first fraction, *overlap*, indicates the regions identified using both selection methods. The average overlap percentage indicates that upwards of 92% of regions have invariant value behavior detected when region formation is guided by either input set. The second fraction, *unique*, designates the regions found only with the second input set. The final fraction, *max*, describes the maximum number of regions found when using both input sets. The average max result indicates that 12% of regions could be lost if only a single input input is used to identify computation regions.

Benchmark	Regions (Train)	Regions (Ref.)	Overlap	Unique	Max
008.espresso	148	155	0.85	0.20	1.20
072.sc	70	71	0.95	0.06	1.07
099.go	440	484	0.80	0.30	1.30
124.m88ksim	128	128	0.99	0.01	1.01
129.compress	36	36	0.98	0.02	1.04
130.li	57	60	0.95	0.10	1.12
132.ijpeg	60	58	0.92	0.05	1.05
147.vortex	192	199	0.97	0.07	1.07
126.gcc	1764	1905	0.86	0.22	1.22
lex	51	53	0.99	0.05	1.06
yacc	69	73	0.96	0.10	1.10
mpeg2	83	80	0.95	0.01	1.02
pgp	51	54	0.82	0.24	1.24
average	-	-	0.922	0.11	1.12

Table 2: Profile-guided region formation statistics.

An important aspect of utilizing run-time value invariance within the region identification process is the assignment of compile-time thresholds. Instruction-level profiling information is used to find individual repeating instructions and to construct large regions of potential reuse in a bottom-up fashion. An instruction is reusable if a percentage of execution is dominated by recurring source operands or access to infrequently changed memory locations, respectively called the *instruction reuse* (R_i) and *memory reuse* (R_m) thresholds. These thresholds indicate the general repetition of instructions using a 16-element history to record the most recent unique computations. Empirical evaluation found that setting R_i and R_m to .65 produces good instances of reusable computation for the base CCR approach.

Code regions may have execution periods in which reuse dominates execution but does not largely account the overall region behavior. Since the DCMS can selectively activate regions to extract performance from such regions by monitoring run-time behavior, the setting of the favorable thresholds changes. *Region layering* is used to grow regions by incrementally lowering the reuse threshold. First, regions with a higher threshold are exposed and then the formation process gathers more regions by steadily relaxing the instruction inclusion threshold. Figure 9 shows the variation in percentage of total dynamic program execution captured in regions by layering with four reuse thresholds: 65%, 60%, 55%, and 50%. Layering establishes regions having good reuse potential (high thresholds) and regions more likely to have periods of reuse (lower thresholds) rather than sustained reuse. The results of indicate that 15% more program execution can identified by lowering the reuse threshold.

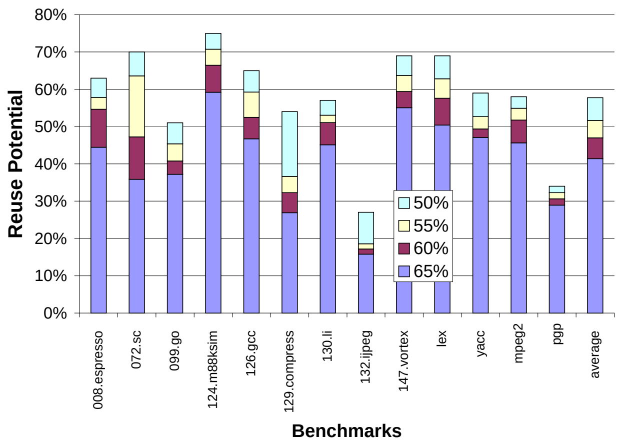

Figure 9: Reuse potential based on region layering.

3.2 Static Computation Region Formation

Although the benefits of profile-directed optimization have been widely accepted, there are several fundamental drawbacks to profiling. First, profiling can be time consuming. Second, profiling may not be feasible in some environments, such as real-time or embedded applications. Third, profiling assumes that program behavior remains relatively constant for all possible inputs. If the program's behavior varies, poor performance after compilation may occur for some inputs. Finally, it is generally infeasible to generate all of the inputs to accurately model all program behavior.

An alternative to using value-invariant profile information is to use static program analysis to find traces of code with invariant behavior. Two methods, *inferred* and *structured* computation region identification are proposed. Both techniques are based on branch execution profiling, a generally accepted technique used in most modern optimizing compilers. Traditionally, to expose sufficient instruction-level parallelism (ILP), basic blocks are coalesced to form *superblocks* [9], which reflect the most frequently executed paths through the code. Superblocks have a single entrance and represent paths with high potential of reuse behavior. Since branches are controlled by program data, the nature of the flow of control through a frequently executed path directly relates to the value locality being exercised by the code's decision components. Hardware concepts proposed to reorder code blocks and store them into a special cache called the trace cache [14] are possibly identifying value-invariant behavior manifesting as frequently executed paths. Such paths, found statically or dynamically, also represent fundamental opportunities for reuse since the each path is generally a long sequence of instructions.

The static region techniques assign superblocks as computation regions for the CCR approach. Candidate superblocks are determined by examining two features: instruction characteristics and region input/output requirements. First, superblocks are selected and partitioned based on the instruction characteristics in the main path of basic blocks. Basic blocks with procedure calls and un-resolvable memory accesses are not included. Candidate regions only include load instructions which have been analyzed as *determinable*, indicating that all potential store instructions to a load can be determined at compile time. The second requirement of candidate superblocks relates to the design of CRB entries. The base model of the CRB supports region entries with an input and output eight-entry register array, storing a computation with mapping between eight input registers and eight output registers. Experiments revealed that 90% of superblocks matched these requirements. Figure 10 illustrates the percentage of program execution for the training input attributed to superblock and candidate superblock traces. These results indicate that a significant percentage of program execution is attributed to candidate regions and can be exposed without value profile information.

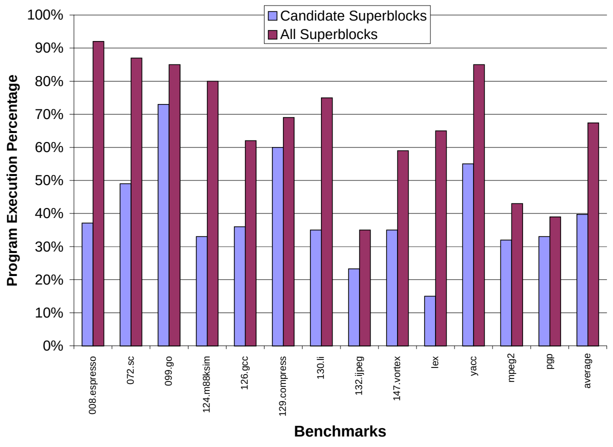

Figure 10: Distribution of dynamic execution in candidate superblocks and all superblocks.

3.2.1 Inferred Computation Regions

The *inferred* region identification technique uses interprocedural propagation of information to identify regions that possess some invariant behavior. The technique uses a coarse-grain dataflow analysis technique to infer value invariance for candidate regions. Inferring invariance in candidate superblocks is based on detecting *constructive* and *destructive* inferring instructions. Constructive inference indicates that a confined set or range of values may occur for an instruction or an instruction operand during some mode of execution of a program. Essentially the technique attempts to locate any value information that may occur at run-time execution. By detecting a set of values or an invariant behavior for a particular variable, the technique can discover a potential region that may have reuse behavior during some input set or program mode. The DCMS enables the activation of such regions when reuse behavior occurs and otherwise prevents the region from attempting reuse.

Memory inferences are made with the support of context-sensitive, flow-insensitive interprocedural alias analysis [6].

Read-only load instructions are candidates for computation reuse since the accessed data is guaranteed to be identical to that of previous references. In addition, write-once data is a common inference related to run-time invariant variables and can be determined by analyzing the callgraph of procedures. To identify write-infrequent inferences of program phases, a related technique for automatically identifying invariant variables, Glacial Variable Analysis (GVA) [2], is extended to conservatively analyze recursive programs. Write-infrequent data is detected when a load is defined at a significantly higher stage (level) in the program's loop-nested annotated callgraph than the respective referencing store instructions. Inference of register operands is based upon interprocedural propagation of value relations throughout the program. The value relation information of expressions is propagated in forward dataflow manner, interprocedurally on the program callgraph and intraprocedurally on each program function. Conversely, the inference system only attempts to determine which value relations are available to an instruction operand during execution. To do this, the inference system modifies existing techniques [4, 17] to propagate value relations until a fixed point is reached.

Destructive program inferences reduce the probability of observing a frequent, but small, number of input varieties. The primary source of destructive inference are sequenced operands, such as loop increment variables. Increment variables used at a nesting level deeper than their definition are constructive inference since their invariance is based on the iteration space of the inner loops. Table 3 illustrates the result of inference analysis on candidate superblocks. The *inferred percentage* is the percentage of instructions with constructive inferences relative to the total number of instructions in the superblock. The data indicates the amount of program execution in superblock regions with the respective inferred percentage. Selection of candidate superblocks is generally best when a high percentage of constructively inferred instructions are located in regions. The results of Table 3 indicate that the inferred percentage of 40-60% enables the majority of candidate superblock execution.

	Inferred Percentage					
Benchmark	0%	20%	40%	60%	80%	100%
008.espresso	42	4	7	25	11	11
072.sc	40	9	9	21	10	10
099.go	20	36	12	19	7	5
124.m88ksim	13	17	4	50	5	11
126.gcc	21	17	21	20	9	10
129.compress	32	18	18	16	8	8
130.li	30	14	12	19	10	16
132.ijpeg	33	26	4	20	8	8
147.vortex	35	6	14	19	13	13
lex	48	3	0	24	12	12
yacc	48	3	1	24	12	12
mpeg2	37	2	24	18	9	9
pgp	35	13	14	20	9	9
average	33.4%	12.9%	10.8%	22.7%	9.5%	10.3%

Table 3: Inferred reuse in superblock execution.

3.2.2 Structured Computation Regions

Some code regions represent fundamental algorithmic computation in which data is manipulated from input sources without any inferred relations. To form regions structurally, two features of each superblock are examined: size and dependence height. Large size is the primary superblock selection constraint, since reusing the results of large superblocks could provide significant performance improvement. Likewise, a large reduction in latency may be achieved if superblocks with a considerable dependence height are reused. In selecting superblocks, these features are referred to as the *structure size* and *structure height* parameters. In addition to superblocks, the structured technique also selects inner loops as candidate regions. Such loops often represent linked-list traversals and array scans that result in significant amounts of redundant execution.

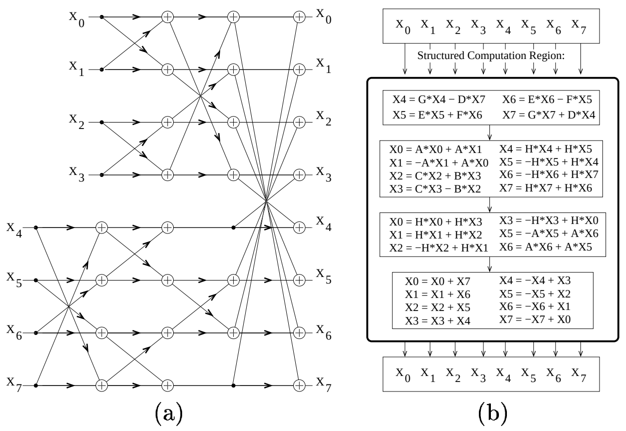

Figure 11: Structured region identification, MPEG2 IDCT dependence graph (a) and computation (b).

Figure 11 illustrates a structured region formed from the row transform of an MPEG2 Inverse Discrete Cosine Transform (IDCT). Figure 11(a) depicts the dependence graph from the Chen IDCT algorithm [5], which requires 36 multiplications and 26 additions and has a dependence height of seven. Figure 11(b) shows the instructions of the dependence graph. These instructions constitute a significant opportunity to eliminate redundant execution due to the inherent nature of the program. Typically, DCT blocks of MPEG-compressed video sequences have only five to six nonzero coefficients, mainly located in the low spatial frequency positions [18]. This property has been used to streamline the design of custom low-power IDCT systems which indicates that the locality is quite persistent. In fact, significant reuse locality (80%) occurs with 16 computation instances.

3.3 Computation Region Summary

Table 4 presents the region count numbers comparing the characteristics of inferred regions and structured regions to profile-guided regions (using a 65% reuse threshold). Overlap occurs when greater than 90% of the instructions of the static regions are found within a profile-guided region. The results of Table 4 show that while the inferred method (using an inferred threshold of 25%) identifies only 49% of the profile-guided regions, using a region height of four and region count of six identifies 68% in the structured approach. Other experiments indicate that together, both static approaches can identify an average of 50% more execution time in regions compared to profile-guided regions.

4. EXPERIMENTAL EVALUATION

The IMPACT compiler and simulator were enhanced to support the proposed architecture framework and the region

	Region Count					
	Inferred			Structured		
Benchmark	Over.	Unique	Max	Over.	Unique	Max
008.espresso	0.70	0.99	1.99	0.72	1.46	2.46
072.sc	0.66	1.05	2.05	0.96	0.17	1.17
099.go	0.53	2.92	3.92	0.88	0.78	1.78
124.m88ksim	0.79	0.35	1.35	0.86	0.26	1.26
126.gcc	0.85	3.24	3.34	0.54	12.50	12.60
129.compress	0.21	0.37	1.37	0.71	0.16	1.16
130.li	0.32	2.04	3.04	0.42	2.33	3.33
132.ijpeg	0.37	0.68	1.68	0.46	0.74	1.74
147.vortex	0.25	0.60	1.60	0.28	0.64	1.64
lex	0.53	2.75	3.75	0.67	2.45	3.45
yacc	0.48	2.03	3.03	0.52	2.24	3.24
mpeg2	0.28	1.37	2.37	0.96	0.08	1.08
pgp	0.45	3.06	4.06	0.95	0.29	1.29
average	0.49	1.65	2.58	0.68	1.85	2.78

Table 4: Static region identification summary.

formation techniques. The processor modeled can issue in-order six operations up to the limit of the available functional units: four integer ALUs, two memory ports, two floating point ALUs, and one branch unit. The integer operations have 1-cycle latency, and load operations have 2-cycle latency. 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 two-level GAs prediction (12-bit history, 16 tables), and a branch misprediction penalty of eight cycles.

The DCMS was configured to the hardware assignments and parameters listed in Table 5. Because the design space is complex, evaluating the individual effect of each hardware parameter was infeasible. Initial parameters were selected and optimal settings were selected based on hardware cost and performance constraints. The RS structure is configured to allow regions with an execution percentage ranging from 6% (32 executions/reset interval) to 3% (16 executions/512 branches) to become candidates. The EB hardware is configured to activate regions with more than 66% (2:1) active intervals and determine an active interval by reusing results greater than half (active threshold percentage) the executions of an interval of its 32 region executions. The RMS parameters are established identically to components in the EB except given slightly more cautious failed activity vectors to readily adapt to changes in reuse requirements. CRB revisions required 12 cycles, representing the migration of a large computation entry to a smaller entry.

Performance and Accuracy. The overall cycle-time speedups for evaluating the reference input set in the CRB and DCMS approaches are presented in Figure 12. The training input set is used in guiding computation region formation. The CRB design is evaluated with a direct-mapped and 2-way set associative mappings. The DCMS models are evaluated with a 32-entry CRB with four equal segments of computation instances respectively holding 2, 4, 8, and 16 computations instances. The two DCMS models are executed with different profile-guided reuse thresholds: 65% and 55%. Performance is reported as speedup relative to the base architecture without CCR support.

Figure 12 is able to illustrate the value of the DCMS approach for two reasons. First, the DCMS approach achieves higher performance due to better computation management

System	Component	Setting
Reuse Sentry (RS)	Number entries	256
	Associativity	2-way
	Exec counter size	5 bits
	Minor Candidate threshold	16
	Major Candidate threshold	32
	Observation reset interval	512 executions
Evaluation Buffer (EB)	Number entries	8
	Associativity	Fully associative
	Exec and miss counter size	5 bits
	Active interval counter	4 bits
	Activity vector size	15 bits
	MinActive vector position	4th active position
	MaxActive vector position	8th active position
	MinFailure vector position	4th failed position
	MaxFailure vector position	6th failed position
	Evaluation reset interval	1024 executions
Reuse Monitoring System (RMS)	Number entries	Size of CRB
	Associativity	Fully associative
	Monitoring reset interval	2048 executions
	Exec and miss counter size	5 bits
	Active interval counter	4 bits
	Activity vector size	13 bits
	MinActive vector position	4th active position
	MaxActive vector position	8th active position
	MinFailure vector position	2th failed position
	MaxFailure vector position	4th failed position
	Active history	8 bits
	Dominant Instance Counter	5 bits
Computation Reuse Buffer (CRB)	Number entries	32
	Number segments	4
	Segment types	2,4,8,16 instances

Table 5: DCMS hardware configuration.

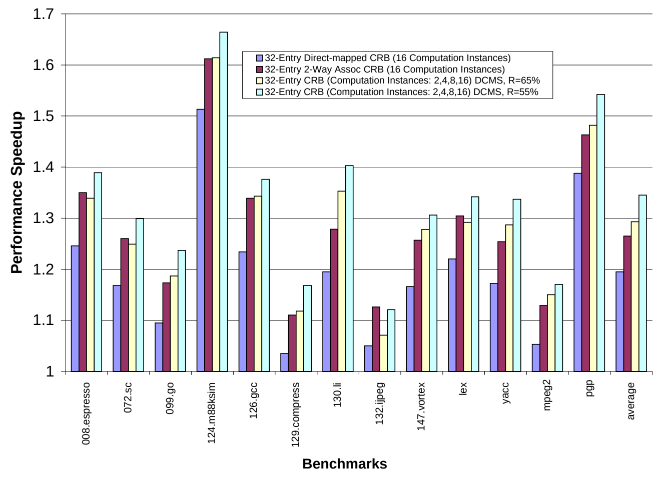

Figure 12: Performance for CRB and DCMS.

properties than the set-associative CRB approach. Essentially the components distributed in the RS, EB, and RMS are collecting run-time information that can improve the ability to manage the CRB better than simply providing more available resources to the CRB. And second, on average, a processor implemented for CCR with DCMS has enhanced speedups due to enabling the reuse in regions not selected for the base CRB modes. The DCMS model with the additional regions selected by lowering reuse identification to 55% is able to provide the DCMS with more opportunities to exploit dynamic computation redundancy. Other experimental results show that on average the performance speedup degrades to 15% if the regions formed with reuse threshold 55% are executed on the set-associative CRB. Generally, the performance benefits of the DCMS approach are enabled by reducing the percentage of failed reuse attempts and increasing the percentage of successful reuse attempts. Figure 13 presents the distribution of reuse attempts for the DCMS relative to the base CRB approach for the lower reuse identification threshold. On average there is both a reduction in reuse failures (22%) and an improvement in successful reuse attempts (21%). The RS and EB stages of the DCMS are working in a coordinated fashion to make only confident reuse attempts.

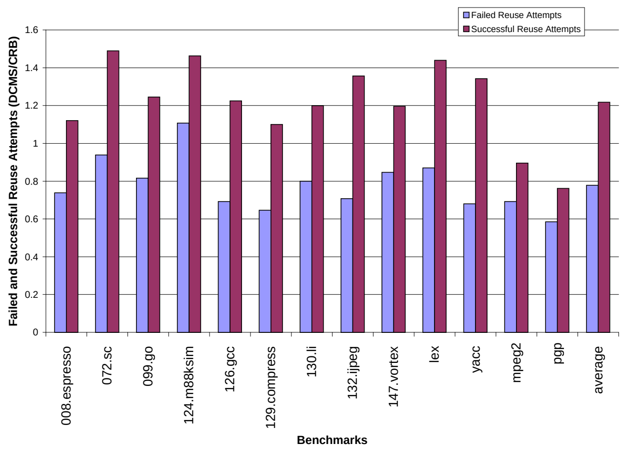

Figure 13: Comparison of failed and successful reuse attempts for the CRB and DCMS models.

Region Formation Evaluation. Figure 14 illustrates the performance speedup achieved using the inferred and structured approaches relative to the profile-guided approach. A third experimental method combines the maximum number of regions found using both static approaches. The results indicate promising potential in eliminating use of value-invariant profile information in the DCMS. On average, the individual static approaches are able to achieve around 40-43% of the speedup of the profile-guided methods. The structured approach has substantially better performance for benchmarks *129.compress*, *mpeg2*, and *pgp*. These programs exploit value-invariant behavior directly related to data input and their computation regions were more readily found using the structure technique. Conversely, the inferred approach is able to better identify regions in benchmarks *124.m88ksim* and *126.gcc* which have abundant inference information. The combined approach significantly improves the percentage of profile-guided speedup achieved to nearly 55%, indicating that the best region identification must coordinate multiple methods.

Another important aspect of the DCMS approach investigated was the evaluation of the region formation techniques on an untrained input. Figure 15 illustrates three methods of region identification: combined profile-guided of the training and reference inputs, combined of the structured and inferred static approaches, and a combined of the profile-guided and static approaches. All methods generate the maximum number of regions corresponding to the multiple inputs or analysis techniques. The results indicate the continued success of both the profile-guided and static

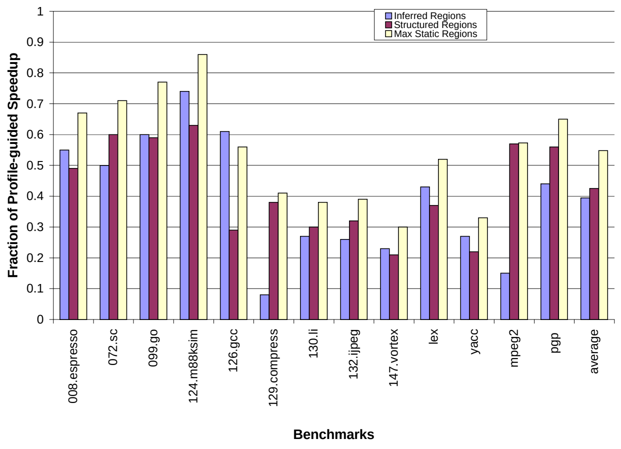

Figure 14: Static region identification performance.

region identification approaches, each respectively averaging 28% and almost 20%. The combined method that employs regions identified using both profile and static methods achieves nearly 5-6% greater performance improvement than the profile-guided method. This result concludes that the profile-guided inputs are not indicating all of the potential program reuse behavior. Thus, only by using static region formation techniques and the DCMS, can the full potential of the CCR approach be achieved.

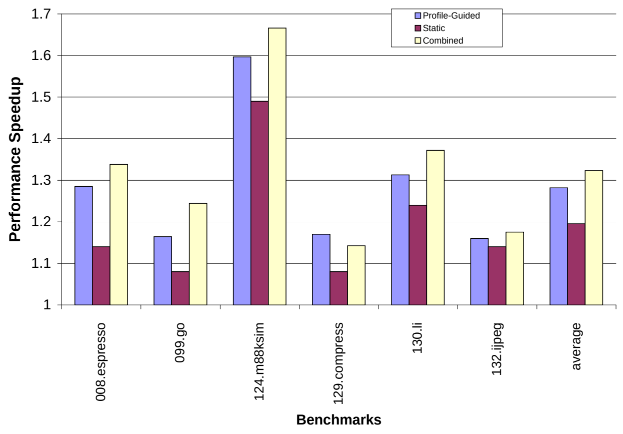

Figure 15: Performance for combining profile-guided and static region identification methods.

Hardware Implementation Costs. Overall, the moderate performance speedups reinforce the strategy of using dynamic management support since the hardware requirements are significantly less than the costs of the base CRB model. Most of reductions comes from having a smaller, but more utilized, CRB. Table 6 represents a hardware cost estimate of the CRB and DCMS models evaluated. This table accounts for bytes of hardware memory, but does not include wiring or logic gate costs. Using this examination, the DCMS requires nearly half the hardware of the base CRB, yet provides improved region management functionality.

Scheme	Component	Cost Expression (in bytes)	Cost
CRB	-	$Num\_Entries * (CI\_entry\_cost(Num\_Instances))$	42624
DCMS	CRB	$Num\_Entries\_Segment[0] * (CI\_entry\_cost(Num\_Instances\_Segment[0])) + ...+$ $Num\_Entries\_Segment[Segments - 1] * (CI\_entry\_cost(Num\_Instances\_Segment[Segments - 1]))$	19980
	RS	$Num\_Entries * (Tag + Exec_{cntr} + State + Target_{addr} + CRB\_index)$	2240
	EB	$Num\_Entries * (Interval_{cntr} + Exec_{cntr} + Miss_{cntr}$ $+Activity_{vector} + Instance\_Indicators(Num\_Instances))$	5276
	RMS	$Num\_Entries * (Interval_{cntr} + Exec_{cntr} + Miss_{cntr} + Activity_{vector}$ $+Activity_{history} + Dominant\_Instance_{cntr} * Num\_Instances\_in\_CRB\_entry)$	316
	(total)	-	27812

Table 6: Hardware cost expressions and cost for models evaluated.

5. SUMMARY

Innovations in high-performance system design and the availability of silicon resources have allowed modern processors to analyze run-time program behavior to effectively manage resources. The DCMS enhances compiler-directed computation reuse by examining reuse execution behavior and dynamically allocate reuse buffer resources. The system selectively deploys code regions for optimal elimination of dynamic computation redundancy. Results show that with little additional hardware, the combination of new region identification techniques and a dynamic management system achieve performance improvement over the traditional computation reuse framework. There is a significant amount of future work to investigate. Further developments in system design include exploring allocation concepts to achieve the optimal tradeoff between power consumption and performance for active computation regions.

6. REFERENCES

[1] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, volume 31, pages 149–159, June 1996.

[2] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. *International Journal of Parallel Programming*, 26(1), February 1998.

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

[4] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the Symposium on Compiler Construction*, 1986.

[5] W. H. Chen, C. H. Smith, and S. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on Communications*, COM-25:1004–1009, September 1977.

[6] B. C. Cheng and W. W. Hwu. Interprocedural pointer analysis using access paths. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[7] D. A. Connors and W. W. Hwu. Compiler-directed computation reuse (CCR). In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 158–169, November 1999.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 19th International Conference on Software Engineering*, pages 213–224, May 1999.

[9] Hwu. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[10] C. Lee and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[11] D. C. Lee, P. J. Crowley, J. L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on windows nt. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 27–38, June 1998.

[12] M. H. Lipasti, C. B. Wilkerson, 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] M. C. Merten, A. R. Trick, and W. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 1999 International Symposium on Computer Architecture*, pages 136–147, May 1999.

[14] E. Rotenberg and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 24–34, December 1996.

[15] 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.

[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] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *Proceedings of the 12th Symposium on Principles of Programming Languages*, pages 291–299, January 1985.

[18] T. Xanthopoulos and A. Chandrakasan. A low-power IDCT macrocell for MPEG-2 exploiting data properties for minimal activity. *IEEE Journal of Solid-State Circuits*, pages 693–703, May 1999.