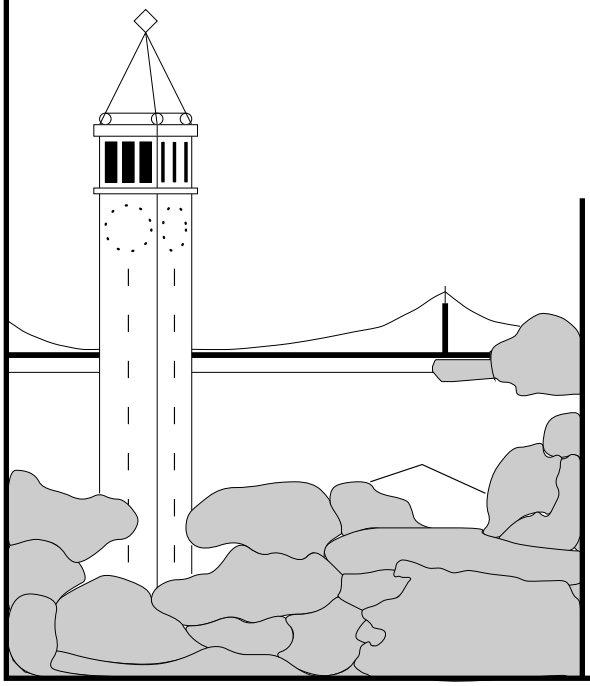
Empirical Study of Opportunities for Bit-Level Specialization in Word-Based Programs

Eylon Caspi



Report No. UCB//CSD-00-1126

January 2001

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Supported by:
DARPA (25502-23797-44-X-NDJCW),
California MICRO (19907-23797-44-X-NDJCW),
Xilinx, Inc.,
Hewlett Packard, Inc.

Abstract

A majority of programs in use today are written for word-based computing architectures, such as the microprocessor, using word-based programming languages. The word model, while convenient, typically provides quantized word widths that are a mismatch for many applications. Consequently, many bits of a word may go unused and contribute no useful information to the computation. Removing these bits from the computation, *e.g.* using specialized hardware data-paths, may provide the implementation with significant savings in run-time, area, and/or power. In this project, we analyze and quantify this bit-level waste using a model of bit constancy and binding-time. Applying the model to the UCLA MediaBench suite [29] of C programs, we find that some 70% of bit-level read operations are to easily identified constant data, much of it in unused, high-order bits. These findings suggest that there is significant opportunity for bit-level specialization of these programs by relatively simple means such as narrower data-paths.

Contents

List of Figures

List of Tables

Acknowledgements

Many thanks to all the people who helped and encouraged me along this project. Thanks to my advisor, John Wawrzynek, for your wisdom in telling me where to start and where to finish. Thanks to André DeHon for your intimate participation in this project and for being like a second advisor to me. Thanks to professor Kurt Keutzer for helping me to see the bigger picture and to place this project in context. Thanks to the office crew (Nick, Randy, Michael, Joe, and all the rest who have come and gone) for pushing me along with reverse psychology. It worked! Special thanks to Denise, my one and only, for patience on the many late nights and weekends. Thanks to my kids, Josh and Ross, for giving me perspective. Thanks to my parents for helping me get to Cal. And thanks to everyone at Cal who has made this one of the greatest adventures of my life.

Chapter 1

Introduction

A large fraction of computation in use today is oriented for word-based languages and architectures that use bit-parallel, SIMD ALUs. The word model is especially convenient for arithmetic and bit-parallel logic, where a single word operation can represents 32 or 64 parallel bit operations. However, the fixed-width nature of the word model is a mismatch for certain computations, *e.g.* boolean logic decisions and arithmetic with small numbers. In such cases, a substantial number of bits in a word contribute no useful information to the computation. In addition, there is evidence [42] [45] [13] [6] that a significant fraction of word operations on traditional word-based architectures are continually repeated with previously-seen inputs, contributing no useful work in steady state. These observations suggest that the traditional word model, despite its convenience, lends itself to significant computational waste, and consequently to degraded run-time, area, and/or power in any implementation.

Existing architectures have sought to overcome wasteful behavior of the word model in several ways. In microprocessors, wasteful word operations can be reduced by value prediction hardware [32] [42] [13] [45], as well as by value-specific optimizations (VSO) using run-time partial evaluation and specialization [26] [10]

[30] [19]. Wasteful bit operations can be reduced in part by going to finer-grained architectures. Recent multimedia instruction sets (*e.g.* Vis [28], MMX [36], AltiVec [35]) have done so using narrow-word, word-parallel ALUs. At the finest grain, custom logic (*e.g.* ASIC) would allow bit-level specialization and optimization. Reconfigurable logic devices (*e.g.* FPGA) would further allow dynamic bit-level specialization, *i.e.* adapting the implementation over time. However, the programming tools available for exploiting such fine-grained architectures are presently limited, and the body of programmers comfortable with fine-grained programming (*e.g.* MMX assembly, hardware description languages (HDL), logic schematic capture) is relatively small. Thus, there is continued interest in using high-level, word-model programming even for custom logic. Recent examples for reconfigurable hardware include PRISC [39], Transmogrifier C [14], NAPA C [15] [16], and C for Garp [7].

This project seeks to quantify the bit-level computational waste inherent in word-based architectures. We propose models of bit-level constancy and lexical binding-times that can be used to gauge the overall opportunity for bit-level specialization of word-based programs. We describe an experimental framework to apply these models empirically to dynamic executions of C programs. Applying this methodology to the UCLA MediaBench suite [29], we find that some 70% of bit-read operations are to easily identified constant data, residing primarily in high-order bits of words.

This report is organized as follows. This chapter (Introduction) begins with an overview and summary of the project. The chapter goes on to recount evidence from the literature of computational waste in the word model and introduces our model for quantifying that waste. Chapter 2 (Methodology) describes an experimental methodology to study that waste in existing C programs. Chapter 3 (Results) describes our findings for the UCLA MediaBench suite [29]. Chapter

4 (Discussion) discusses problems and possible extensions to the methodology as well as ideas for exploiting bit-level waste by logic-level specialization. Several appendices are included as a reference for certain details of the experimental implementation.

1.1 Project Overview

One of the benefits of reconfigurable computing devices is the ability to implement data-paths that are highly specialized to an application or to its input data. In particular, because they are programmable, reconfigurable devices have the potential to instantiate and execute only the minimum hardware actually required for each application. For instance, in a fine-grained device (*e.g.* FPGA), it is possible to implement ALU operations (add, subtract, etc.) of arbitrary bit widths. Contrast this with microprocessors, where all operations are done in large, fixed-width ALUs. The computation of unused high-order bits in ALUs is wasteful.

It is also possible with reconfigurable devices to specialize a data-path around known data values and to create a partially-evaluated, hence smaller, data-path. For instance, a multiplier circuit with one constant input can be made substantially smaller and faster than a general multiplier where both inputs are variable [33] [9]. Similarly, random logic can be partially-evaluated using bit-level constant folding (*e.g.* "x AND 1" simplifies to x). This has been demonstrated in a number of applications implemented on FPGAs (*e.g.* SAT solving [47] [38], string matching [20]), where specializing a circuit for particular, known data values made for a substantially smaller and faster implementation (*e.g.* specializing for a particular boolean formula for SAT or a particular character sequence for string matching).

The performance benefit of specialization can be attributed to two effects: smaller computational delay and smaller circuit area. Smaller delay should in

general reduce total run-time. In a programmable device with limited size, using a specialized circuit of smaller area frees up resources to execute additional, parallel operations. Thus, specializing for smaller area can also contribute to smaller overall run-time.

The question posed in this master's project is how much raw opportunity is there for specializing computations? Can one always expect to speed up an application by some factor using data-path specialization? The answer to this question clearly depends on the application– some applications use more constant data than others. The answer also depends on the representation of an application. For example, an application written in a high-level language such as C is forced to use particular data representations, *e.g.* 16 or 32 bit integers, potentially much wider than the application requires. In contrast, an application written in a hardware description language such as Verilog can use arbitrary bit widths.

This study focuses on applications written in C. C is a popular, high-level language that represents a large body of computation in use today. Its word-based, imperative, sequential style of programming represents the way that a majority of designers think about and code applications. Thus it is a reasonable base representation for analysis and comparison in asking how much opportunity for specialization exists. Furthermore, there is a large base of existing code and benchmarks written in C, available for analysis.

Our analysis of opportunity for specialization will necessarily be specific to the program representation considered herein, namely C. Thus we must define what is a "wasteful" operation in this context, and how it can be improved or avoided by specialization. We can define a "wasteful" computation as one whose result produces no new information. This typically happens when the input values of the computation are known, *i.e.* are constant, or have been seen before. Such a computation could be specialized by pre-computing or memoizing the result,

thus avoiding the run-time computation. Because in C the inputs and outputs of computations are represented by named variables and memory locations, this study will concentrate on identifying constant values in variables and memory locations.

At the word level, an example of a wasteful operation with known inputs is multiplication by a constant such as one. The answer is trivially known to be the multiplicand and does not actually need to be multiplied. More generally, the actual value of the multiplicand may not be known until run-time, even though it is known that the value will be constant. This is an example of a dynamic constant, in which case it may be profitable to replace the multiplier by a constant multiplier as soon as the multiplicand value is encountered at run-time. More generally, in a hardware implementation, we must consider operations at the bit-level. That multiplicand may not have a strictly constant value, but it may have certain constant bits. For example, it may always be even (least significant bit is zero), or it may always be smaller than 256 (*i.e.* it has only 8 dynamic, low-order bits). An operation with certain known input bits might be specialized at the bit level, for instance by removing some partial products of the multiplier, or more generally in a hardware implementation, by partially-evaluating arbitrary bit operations (AND, OR, etc.). Hence, in our analysis of variables and memory locations in C, we will identify not only constant words, but constant bits within words.

In searching to expose the total, actual opportunity to specialize around constant values, our analysis must be able to get around those artifacts of the C language that occlude constancy and opportunity. Analyzing the flow of data values is particularly difficult in C, thanks to sequentialization (the fact that a value can be stored in memory and used later in the program) and thanks to pointers (which make it difficult to know where a value is stored in memory and which value is subsequently read from memory). In addition, we wish to account

for data-dependent behavior, *i.e.* for constancy of values that are not known at compile time. Thus, our analysis must be a dynamic, run-time analysis that looks at actual program values and execution paths. A static, compile-time analysis is insufficient.

The hypothesis being explored here is that programs written in C exhibit a significant amount of dynamic constancy, at the bit level. An experimental methodology will be developed to discover those constant bits and to relate them directly to the C source. That is, the methodology will discover constant and unchanging bits in named variables. Furthermore, the methodology will discover how often those bits change with respect to the block-structure of the C program.

A corollary to our hypothesis of the existence of bit-level constancy is that this constancy can be exploited by data-path specialization. Hence, if a C program were to be implemented in hardware, our methodology could pinpoint which C constructs, and in particular, which bits, are candidate for specialization. Because our methodology is based on dynamic, data-dependent program behavior, its results are not necessarily correct for every data set. Hence, its results cannot be used directly to specialize a program without the use of guards, *i.e.* additional error-checking expressions or hardware to signal when an input bit's value is other than the expected constant. Alternatively, the results of analysis can be used to guide a subsequent, static, compiler analysis which would prove, for all possible input data, the veracity of some of the dynamically-discovered constancy.

Finally, we choose the MediaBench suite of programs as an initial target for analysis. The suite is comprised of various media processing tasks, including signal-processing, data compression, and protocol/file processing. One common element in all these programs is that they process large streams of data (*e.g.* audio, input images) using fairly simple-structured, unchanging data-paths (*e.g.* DCT, ADPCM). Because of the mostly static nature of the data-paths, we expect a cer-

tain amount of constancy or repeated work in the computation (*e.g.* multiplication by a fixed DCT coefficient). Furthermore, the data-paths are often parameterized by command-line options (*e.g.* the level of compression in JPEG encoding). These options determine constants and operating modes that remain fixed for the entire duration of program execution. Such behavior is a prime candidate for specialization, wherein it would be necessary to specialize a data-path only once, near the beginning of execution, around constants derived from command-line options.

1.2 Constancy in the Word Model

At the heart of this project is the premise that computation on known inputs produces no useful new information. The results of such a computation could be cached or hard-wired, removing the need to perform the computation at run-time. This notion is by no means new. It is commonly used in modern optimizing compilers, run-time specialization systems, and microprocessor hardware (in various forms of caching and prediction). We are particularly interested in extending this premise to the bit level, *i.e.* that a bit operation with known bit inputs produces no useful new information. Since word operations are implemented by bit operations (ultimately by boolean logic), any word operation on inputs containing known bits will exhibit computational waste in the bit-level implementation.

In this section, we recount evidence of constancy and trivial computation in word-based architectures. We first discuss instances of word-level constancy, *e.g.* computation on known, repeated, or slowly-varying word values. We then discuss instances of bit-level constancy in word-based computation, *e.g.* range-limited arithmetic (where high-order bits are zero) and strided arithmetic (where low-order bits are fixed).

1.2.1 Word-Level Constancy

Traditional compile-time constant propagation and constant folding are useful when the value of constant operands is known to the compiler. The compiler simply replaces the computation by its result, *e.g.* `1+1` becomes `2`, and `x*1.0` becomes x. Other traditional techniques, such as hoisting invariants out of loops, can be applied even to *dynamically valued constants* (DVC) whose exact value is not known at compile-time. Run-time partial evaluation [26] [10] [30] can use all these techniques once a DVC value is known, to create a value-specific, specialized implementation.

Compiler techniques to deal with constancy are complicated by the typical lack or under-use of mechanisms to declare constant and dynamically-constant variables. For instance, Schilling [43] analyzes a collection of scientific and compiler-related C++ programs to find that less than 1% of all variables were declared DVC (using `const`), even though 30-50% of local, non-class variables behaved as DVCs. Perkins [37] finds similarly for 2 million lines of defense-related ADA code, with up to 50% of variables being DVC candidates. A number of program analyses exist to automatically discover DVCs and slowly-varying quantities, including binding-time analysis [25] and staging analysis [27] [1]. Nevertheless, there remain run-time constants that cannot be discovered by practical analysis, for instance due to heavy aliasing through pointers or files.

A weaker form of constancy exists when a word quantity takes on more than one value, but there are very few values, and/or the values change very infrequently. Evidence of this is abundant for microprocessors. Lipasti and Shen [31] report that, across a collection of SPEC and multimedia-oriented C programs compiled for a PowerPC 620, 49% of dynamic instruction executions repeat operands from their previous execution; 61% repeat operands from one of the previous 4 executions. Similarly, Wang and Franklin [45] report that, across a subset of

SPECint92, 39%–79% of dynamic instruction executions repeat one of the previous 16 executions. Sazeides and Smith [42] report that over 50% of dynamic instruction executions generate fewer than 64 values, and over 50% of static instructions generate only 1 value. Calder *et al.* [6] reports that among integer instructions, up to 54% of dynamic instruction executions repeat a single, most frequent value. These cases of operations with previously-seen inputs can be exploited by *data value prediction* hardware [32] [42] [13] [45] which caches and reuses results instead of recomputing them.

Alternatively, operations with particularly simple inputs (*e.g.* arithmetic with a 0 or 1 operand) could be special-cased in logic rather than have their results cached. Richardson [41] reports that up to 6% of dynamic instruction executions in SPEC and up to 7% in the PERFECT Club are arithmetically *trivial*, *e.g.* multiplication by 0 or 1 and division by self. The latency of a trivialized operation can be improved by specializing the operation to recognize and handle special-case inputs, *e.g.* to directly emit 0 for the multiplication $x \times 0$. This latency advantage can be had even if the special-case inputs are not frequent. Nevertheless, there is an area cost for this approach, since it augments rather than replaces the original implementation with a specialized one.

1.2.2 Narrow-Width Computation

Recent microprocessors have moved to wide ALUs, typically 32 or 64 bits, primarily to handle growing address spaces. Typical integer data values, however, have not grown substantially and do not use the full width of these ALUs. The high-order bits of data quantities often remain zero or act merely as sign extension bits. These high-order bits carry no useful information (at most, one bit's worth for a sign), and the ALU bit operations associated with them are effectively wasted. For instance, many signal processing programs use 16- or 24-bit

arithmetic to implement prevailing standards. Also, boolean logic for control-flow decisions typically operates on single-bit quantities. When run on a 32-bit or 64-bit ALU, these cases invoke many wasted bit operations.

Brooks and Martonosi [4] report that, in a 64-bit processor implementation of the SPECint95 and MediaBench suites, some 50% of all dynamic instructions have all their operands no wider than 16 bits. An additional 30%–40% of dynamic instructions use precisely 33 bits for heap and stack addressing. Nearly no instructions use the full 64 bits.

The recent trend in microprocessors to SIMD multimedia operations (*e.g.* Vis [28], MMX [36], AltiVec [35]) represents one solution for using narrow arithmetic efficiently. These architectures segment a wide ALU (typically 128 or 256 bits) into a vector of parallel, narrow operations (typically 8, 16, or 32 bits). Using such architectures efficiently requires vectorizing an algorithm. Furthermore, the bit-width of all operations must be known statically, and for peak efficiency, must be a power of 2. Thus, these architectures can still waste up to 50% of bit operations, for instance in the case of 16-bit arithmetic implemented in 17 bits to handle overflow.

Brooks and Martonosi [4] describe a SIMD processor architecture that alleviates the need for static knowledge of bit-widths by dynamically recognizing 16-bit operands. Selectively gating the clock to the unneeded, upper 48 bits of a 64-bit ALU produced a power reduction in the ALU of just over 50%. Dynamically vectorizing 16-bit operations by packing them 4-wide into the 64-bit ALU produced a 4% speedup for SPECint95 and 8% speedup for MediaBench.

Determining a precise, minimum bit-width for word operations can be difficult or impossible. Conservative lower bounds for data widths can be found using value range analysis [22] [2]. Value range analysis is a program analysis that uses abstract interpretation to estimate the value range of word quantities. For

instance, the sum of $x \in [0, 10]$ plus $y \in [0, 10]$ is $(x + y) \in [0, 20]$. Range-limited quantities typically have many high-order bits being zero, signed, or otherwise constant. There also exist a number of analyses that estimate data widths directly at the bit level [39] [8] [5]. These analyses typically use abstract interpretation based on the logic implementation of a computation rather than on its arithmetic properties. For instance, the sum of two 8-bit values is a 9-bit value. These analyses are conservative and may overestimate the minimum required width of a computation. Furthermore, because they are static and performed at compile time, these analyses cannot discover cases where word values are dynamically narrow at run-time. To find tight bounds on data widths, the approach of this project is bit-level and uses dynamic profiling rather than compile-time analysis.

Finding the most efficient bit-width for a hardware implementation is more complicated than just finding data widths. One approach for improving the computational efficiency of a word ALU (as measured by some metric such as bit-operations per area-time) is to implement narrower ALU *slices* and to chain them to perform wide operations. Chaining may be performed sequentially on one slice or spatially using multiple parallel slices. This approach still suffers from some fragmentation, since in any operation whose width is not a multiple of the slice width, one ALU slice will not use all of its bits. Recent examples include the reconfigurable arithmetic array of Haynes and Cheung [24], comprised of programmable $k \times k$ multiply-add blocks (*Flexible Array Blocks*, or *FAB*s). Under the assumption that the frequency of $n \times n$ multiplies in designs falls off as $1/n$, Haynes and Cheung find the most area-efficient FAB size to be 8×8. Other sliced, ALU-based, reconfigurable systems include CHESS [34] based on 4×4 ALU elements, or at a finer level, Garp [23] based on 2-bit LUT/ALU elements. The most efficient slice width is dependent not only on the statistics of target applications but on the architectural overheads of the implementation. In some cases, single-bit slices

may not be the most efficient.

Bondalapati and Prasanna [3] consider the more sophisticated case of bit widths that grow with time. They describe a *dynamic precision management* scheme where a data-path is periodically reconfigured to the precise width required at that time. Although they target a reconfigurable fabric of single-bit elements, there is nothing in principle that prevents their technique from being applied to fabrics with wider ALU slices.

1.2.3 Address Computation

Address arithmetic is one kind of computation that seems highly prone to bit-level constancy. For one thing, 64-bit pointers in a small or under-used address space will clearly have many zero high-order bits. Brooks and Martonosi [4] also note that adding small offsets to large base addresses is not likely to overflow to the full width of the base address. Thus small offset operations can be implemented using narrow arithmetic, so long as a mechanism exists to detect and handle the overflow case. This speculative approach added a 4% speedup for SPEC in their dynamic vectorization approach.

The alignment of memory structures on word boundaries suggests that there is some constancy in the lower bits of addresses. For instance, stepping through an array of 32-bit words in a byte-addressable memory requires a strided pointer whose lowest 2 bits are always zero. More generally, stepping through an array of 2^n-byte objects, even if every access is offset within the object, will use a strided pointer with n constant least-significant bits. This phenomenon is a special case of strided data patterns, whose redundancy can be alleviated in microprocessor systems by strided value prediction hardware [42] [13]. The project described in this paper finds that the waste due to constant low-order address bits is small in the MediaBench suite.

1.3 Profiling to Discover Constancy

This project uses program profiling to study bit-level waste in word-based architectures. Here we introduce and motivate the salient features of our profiling approach. Details of the experimental methodology are deferred until Chapter 2 (Methodology).

1.3.1 Capturing Dynamic Behavior

To accurately quantify the bit-level waste of a computation, we use a technique of *profiling*. Profiling collects dynamic information from an actual execution of a program. Hence, unlike static program analyses, profiling reveals the actual, data-dependent statistics of a program.

1.3.2 Source-level Profiling in C

We choose to study programs in the C language as representative of word-based architectures. C is a popular, word-based programming language that is relatively low-level. That is, the language directly represents many features of existing, word-based processing architectures, particularly features of microprocessors, including: quantized word widths (*e.g.* 8, 16, 32 bits); scalar data types with known bit-level representation (*e.g.* `signed` is two's complement); and pointer-based memory access. The popularity of C and of its typical target architecture, the microprocessor, suggest that a judicious collection of C programs can be taken as representative of modern, word-based computing. The popularity of C also means that a variety of accepted benchmarks, compilers, and analysis tools are readily available. We concentrate on the UCLA MediaBench suite [29], a collection of multimedia-oriented benchmarks. We use the SUIF C compiler [46] to modify applications for run-time profiling.

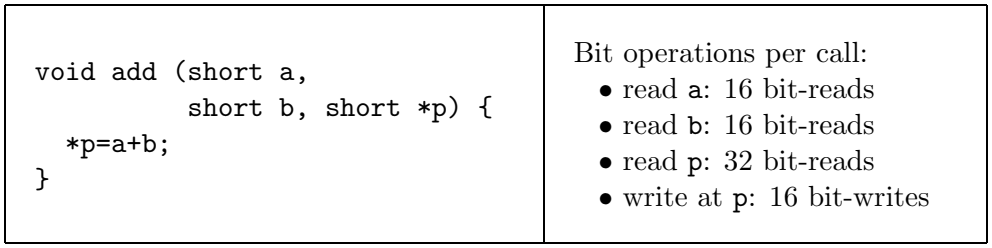


Figure 1.1: Tallying bit-reads in a sample function invocation

Our profiling methodology collects statistics at the source level, *i.e.* on quantities such as named variables that can be related directly back to the program source. This makes the results largely independent of any particular hardware implementation. In tallying bit operations, the profiler considers each word operation to have the minimum bit width mandated by the original program's data types, not the fixed width of any particular ALU implementation. This makes our profiling results more conservative (*i.e.* they report less constancy) than typical microprocessor-oriented profilers that count all 32 or 64 bits of the ALU as active in every operation.

Profiling a program at the source level yields results that reflect bit-level waste in the original, source description of the computation. Normally, this is not the computation that would be executed in hardware. Rather, a program typically undergoes extensive compiler optimizations before execution. Hence, it is preferable to profile program source after it is modified by conventional compiler optimizations. This is possible with the SUIF compiler, which applies optimizations at the source level.[1]

1.3.3 Storage-based Profiling

The profiling methodology presented here targets a program's data storage. It collects statistics on access and modification of variables and heap memory. These

[1]SUIF operates on an internal form that is translatable one-to-one with C source.

storage elements represent the inputs and outputs of primitive operations such as arithmetic and data movement. We use dynamic read accesses as a measure of computational activity, since each read access (*e.g.* of a named variable) is an input to some operation. Moreover, this measure is independent of the particular operation being used. Counting each word read as a collection of *bit-reads* allows us to represent generic ALU bit operations. An example of tallying bit-reads is shown in Figure 1.1. Note that the addition operation `a+b` is accounted for in the reading of its two 16-bit inputs.

A bit of storage is an interesting constant if its value does not change for some duration of time. The storage location may actually be rewritten many times without changing the value of an individual bit. If the bit is read and used many times without changing value, then it may be profitable to specialize around it as a constant. Profiling named variables at the source level allows us to relate the frequency of change of a storage bit to the program structure. For instance, it is possible to determine that a bit of a variable is constant in a particular code block such as the body of a loop. In this way, storage profiling can empirically discover the binding-time of variable bit and identify it as a DVC.

Rather than profile storage, an alternative approach is to profile instruction operands directly. This approach is popular with microprocessors, where execution of instructions with previously-seen operands can be sped up by value prediction hardware [31] [42] [13] [45] [6]. Unlike storage profiling, instruction profiling easily differentiates the constancy behavior of different kinds of operations. However, instruction-centric results are necessarily architecture-specific, tied to a particular processor ISA or a compiler's intermediate form (*e.g.* low-SUIF in the SUIF compiler). Also, instruction-centric results are difficult to relate back to the original program source.

1.4 Model of Bit-Level Constancy

Section 1.2 described a number of different forms of constancy that lead to computational waste in word-based architectures. Word-level constancy includes purely constant words, slowly-varying words, and words that take on a small set of values. Sub-word constancy includes range-limited words with constant high-order bits, strided words with constant low-order bits, and words with an evolving precision requirement.

We now develop a model of constancy that captures most of the above forms of constancy *at the bit-level*. Our model works on the binary representation level rather than focusing on arithmetic word properties such as value range or additive stride. This bit-level approach is oriented for evaluating hardware costs, since it identifies bits in a data-path which are wasteful and might be specialized. Applying the model to a program gauges its overall opportunity for bit-level specialization by such means as narrower data-paths and logic optimization.

Our constancy model uses a notion of bit-level *binding-times*. This idea is based on conventional, word-level *binding-time analysis* [25], a static analysis that assigns to each program quantity a label from the domain: $U \sqsubset S \sqsubset D$ (U=undefined, S=static, D=dynamic). Our model extends this domain in two ways. First, our domain describes individual bits of words, differentiating between sign-extension and non-sign bits. Sign-extension bits represent a particular kind of bit-level waste, since they comprise a sub-word region where all bits carry the same value (all zeros or all ones). Sign-extension bits need not be constant, but they are perfectly correlated and can generally be replaced in hardware by a single representative bit.

Second, our domain fills the gap between the pure static (S) and dynamic (D) classes with additional labels to denote frequency of change. This notion of frequency is related to a program's lexical structure. A bit value may be constant

across all executions of the program, constant only during individual executions of the program, or constant only inside some lexical block. In particular, there is a class for bits of a local variable that are constant in any dynamic entry into the variable's scope of definition. Our model does not identify constancy on any finer time scale, for example within smaller nested blocks. Thus this lexical notion of frequency of change is distinctly weaker than flow-sensitive formulations of classic binding-time analysis, which label a variable with a different binding-time at each point in the program. Also, this lexical notion of frequency is applicable to variables but not to the heap, since heap objects do not have a lexical scope of definition. Thus heap objects are analyzed with only a subset of the bit binding-time domain.

Whereas conventional binding-time analysis gauges constancy statically from definitions (writes) in source code, our bit binding-time model is meant to capture constancy from dynamic execution. Dynamic constancy is concerned only with retention or change of a bit's value. A bit of storage may be rewritten many times during execution yet always retain its same value. An example of this is the unused high-order bits of a range-limited variable, which are always zero as new variable values are written. Such bits are effectively constant, but this property is evident only dynamically, not statically. Our run-time analysis ignores write operations that retain a bit's previous value. In the remainder of this report, the terms bit binding-time and bit constancy may be used interchangeably.

Figure 1.2 shows the domain of *bit binding-times* $\mathcal{B}$. The domain is a lattice with bottom element Undefined and top element Per-Def (equivalent to the bottom U and top D of conventional binding-time analysis). Elements closer to the bottom are more static, whereas elements closer to the top are more dynamic. Strictly speaking, our model does not use this bit domain but rather a word domain built from it. The binding-time domain $\mathcal{D}_n$ for an n-bit word is formed as the

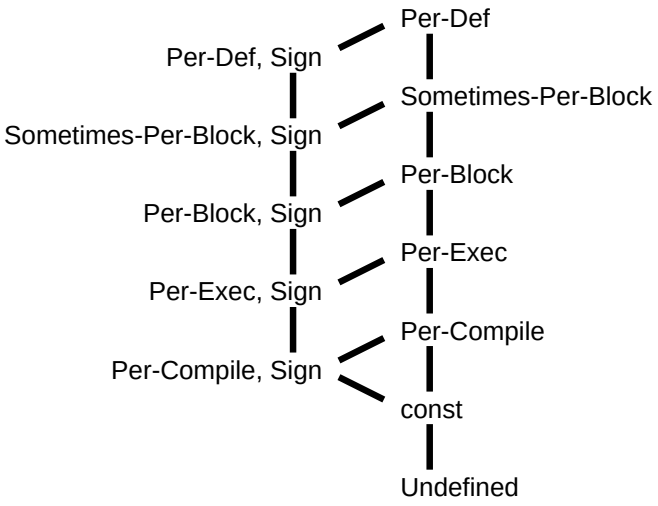


Figure 1.2: The bit binding-time domain $\mathcal{B}$.

direct product of n instances of $\mathcal{B}$, with point-wise (bit-wise) least-upper-bound (LUB), and with the additional restriction that sign extension bits must appear in a contiguous region of high-order bit positions.

The non-sign elements of the bit binding-time domain (shown on the right side of Figure 1.2) are:

Undefined ($\perp$ bottom element) An "undefined" bit is a bit of allocated storage whose value is never defined by a write operation. In practice, we find that read activity to undefined storage is negligible, so we do not report information on undefined bits.

const A "const" bit is a bit from a variable that is defined `const` in the program source.

Per-Compile A bit bound "once per compile" is one that takes only one unique value across all executions of the compiled program.

Per-Exec A bit bound "once per execution" is one that takes only one unique value during an execution of the program. That value may be different in different executions.

Per-Block A bit bound "once per block" takes only one unique value during any entry into its variable's scope of definition. This class applies to local variables that are instantiated anew every time control enters their scope of definition. A bit bound "once per block" is a DVC in that its value is constant during any scope instantiation of its variable, but the value may be different in different instantiations.

Sometimes-Per-Block A bit bound "sometimes per block" takes only one unique value during *some* scope instantiations of its variable. The bit takes multiple values in at least one scope instantiation.

Per-Def ($\top$ top element) A bit bound "once per definition" is a dynamic bit whose pattern of change is not simply related to its variable's scope of definition.[2] This is a catch-all class and is necessarily imprecise. The name "once per definition" is taken for symmetry with other binding-times and does not necessarily imply a new unique value at *every* definition point.

The sign-extension elements of the bit binding-time domain (shown on the left side of Figure 1.2) designate bits that serve as sign extensions in their respective words. Sign extension bits always appear in a contiguous region of identically-valued bits (all zeros or all ones) in the high-order positions. If these bits change, they change together. The sign-extension elements of the bit binding-time domain correspond to the non-sign elements in frequency of change. These elements include:

Per-Compile, Signed
Per-Exec, Signed
Per-Block, Signed
Sometimes-Per-Block, Signed
Per-Def, Signed

[2] A bit in the Per-Def class may be slowly varying in a way that does not match the model. It is possible for a dynamically constant bit to be labeled Per-Def instead of Per-Block if its scope of definition is much larger than its scope of use. A solution to this is described in Section 4.3.2.

The bit binding-time domain is restricted for bits on the heap, since heap objects have no lexically-oriented lifetimes. The domain is restricted to the elements: undefined, Per-Exec (*i.e.* static), Per-Def (*i.e.* dynamic), and their signed counterparts. This essentially reverts the domain into a bit-level version of the classical binding-time domain: $U \sqsubset S \sqsubset D$, with no continuum between the static and dynamic extremes.

The domain described herein captures all the forms of constancy described at the beginning of this section except for one. It cannot represent a dynamic word that takes on a small set of arbitrary values. It can, nevertheless, represent a small value set that is range-limited, *i.e.* has constant or sign bits in the high-order positions. More generally, the domain can conservatively represent a small value set whose values differ in only a few bits. The like bits would be designated constant, while differing bits would be designated dynamic.

A bit's binding-time is discovered in profiling by a method of progressive refinement. Each bit of allocated storage begins with the conservative designation undefined. During execution, as the bit value is defined and redefined by write operations, its binding-time designation moves up the domain lattice via least-upper-bound (LUB) operations. The mechanism is detailed in Chapter 2 (Methodology) and described only briefly here.

The profiling system maintains at run-time a global, incremental binding-time for each bit of allocated storage. For bits of variables, each scope instantiation of the variable yields a local binding-time based on whether the bit was changed during that instantiation. At the exit from each instantiation, the profiling system updates the global binding-time for the bit by LUBing it with the local binding-time. Bits of heap objects have only a global, incremental binding-time, since they are not associated with lexical blocks. That binding-time is updated by LUB every time the heap bit is written. The actual binding-time of a bit is

taken as its global, incremental binding-time at program exit. Bits of storage that are never written simply retain their undefined designation throughout execution. Such bits are intentionally omitted from the result report.

Chapter 2

Methodology

To study bit constancy in a particular C program, we modify the program into a self-profiling version. At run-time, the profiling code processes all read and write accesses to variables and heap storage. The profiling code computes bit binding-times incrementally, as the program runs, and emits a result report at program exit. This methodology does not emit memory traces and does not require any post-processing after program termination.

This chapter describes each phase of the self-profiling methodology: instrumenting a program's source code, compiling, processing statistics at run-time, and forming the result report. In addition, this chapter describes *call chain disambiguation*, a program transformation that can be applied before instrumentation to make the analysis context-sensitive (*i.e.* to yield separate results for a variable depending on how its scope is reached in the call chain).

2.1 Overview

To profile a program, we instrument (modify) it at the source level to profile every read and write access to variables and heap storage. The instrumentation process involves enumerating all named variables and all lexical blocks, then inserting

library calls into the code at every event of interest. The calls are to profiling routines that collect and compute statistics as the program runs. The instrumentation process is described in Section 2.2. The actions of the profiling routines are described in Section 2.3.

The binding-time of each storage bit is computed incrementally as the program runs. For each bit of storage, the profiling code maintains a *memory image* of the bit's most recent value, along with an incremental binding-time, access counters, and other information. Value changes are detected at write operations by comparing the written value with the most recent value.

For local variables, which are instantiated anew at each entry into the variable's scope, the profiling code maintain a separate, local memory image for each scope instantiation. This local information is merged into the variable's global memory image upon exit from the variable's scope block. Local memory images are kept in a stack to support recursion. In this manner, the binding-time of a variable bit can incorporate information from all local instances of the variable. Note that the management of local information requires instrumenting the entry and exit into each program block, with special handling for abnormal entry and exit via `goto` or `return`.

We make no attempt to associate heap storage with any lexical scope. There are no local memory images and no assessment of the frequency of change. In effect, heap bits are analyzed with only a subset of the bit binding-time domain (Undefined, Per-Exec, Per-Def, and the signed variants thereof).

The present implementation does not identify Per-Compile bits. Determining that a bit is bound Per-Compile would require comparing its value across multiple executions of the program. Values would have to be stored in a file between executions and read-in for comparison in each new execution. For the sake of simplicity, we avoid these mechanisms altogether and concentrate only on individual

executions. The strongest binding-time that the implementation can identify in a single execution is Per-Exec.

2.2 Instrumentation

Instrumentation of programs is done using the Stanford SUIF C compiler[1] [46]. SUIF is a freely available, extensible, optimizing, ANSI C compiler.[2] Though SUIF has few processor-specific back-ends, its internal program representation can be readily converted back into C for compilation on any target processor. We use SUIF in this way as a source-level transformation engine.

There are a number of program transformations that can be performed prior to instrumentation to enhance the profiling results, including a variety of traditional compiler optimizations. In this section we discuss the transformations and instrumentation that convert a program into a self-profiling version.

2.2.1 Program Transformations

There is just concern that profiling a program at the source level may not reflect the operations actually performed in hardware. Hence, we apply a number of program transformations prior to instrumentation to help expose the actual operations of a hardware implementation.

Profiling an unoptimized program may discover unrealistically plentiful constancy that is normally reduced by compiler optimizations. Hence, we apply traditional compiler optimizations in SUIF before instrumenting a program. Traditional optimizations such as constant propagation and folding, common subexpression elimination, and hoisting loop invariants, can reduce word-level waste

[1] We use SUIF version 1 and the Harvard MACHSUIF extensions [44] for bug fixes.

[2] Following ANSI C, SUIF considers `int` integers to be 32-bits. Hence, our results on bit utilization are less aggressive than studies that consider large, 64-bit ALUs.

by removing ALU and memory operations and possibly by removing variables altogether.

The compiler optimizations performed using SUIF (`porky`) are listed below. Appendix A gives more details on each optimization from the `porky` man page.

- `-forward-prop` — forward-propagate calculation of variable definitions to the point of use
- `-const-prop` — constant propagation
- `-dead-code` — dead code elimination
- `-privatize`, `-glob-priv` — privatize global variables into locals
- `-cse` — common subexpression elimination
- `-loop-invariants`, `-loop-cond` — hoist invariants out of a loop
- `-iterate` — iterate optimizations until they converge

Several other useful optimizations offered by SUIF were not used because they proved to be too buggy:

- `-fold` — constant folding
- `-reduction` — reduction to move summation out of a loop
- `-ivar` — induction variable detection and reduction

These and other optimizations may become available and reliable in future versions of SUIF, for instance in SUIF2 (in beta November 1999).

Profiling read accesses to variables and heap objects may not fully capture the computational complexity of deep, nested expressions. Since storage-based profiling ignores operations that do not access storage, it would ignore many intermediate operations in a deep expression tree. For instance, profiling the code of Figure 2.1(a) would capture the two additions, since their inputs read variables,

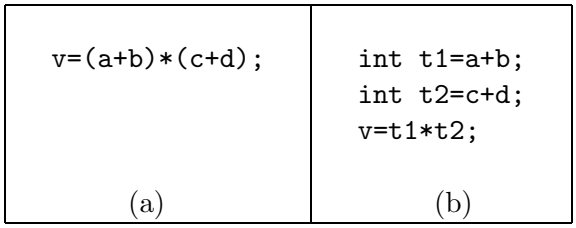


Figure 2.1: Exposing intermediate values as temporary variables

but not the multiplication. In effect, storage profiling captures only the dynamic inputs and outputs of an expression tree. One way to better capture the complexity of a deep expression tree is to flatten it and to expose all intermediate quantities as temporary variables. For instance, profiling the transformed code of Figure 2.1(b) would capture the multiplication, since its inputs now read temporary variables. Unfortunately, such a flattening transformation is likely to vastly expand a program's code size and run-time, hence it is *not* used in the present implementation.

Section 2.5 describes a program transformation (*call chain disambiguation*) that adds context sensitivity to the profiling analysis. A context sensitive analysis would identify different results for a particular variable depending on how its scope block is reached in the call chain. The proposed transformation is a brute-force approach. It disambiguates a variable's context by uniquely duplicating procedures along the call chain.

2.2.2 Instrumented Events

Here we describe all events in a program's source code that are instrumented. Most events are instrumented by the insertion of calls to run-time profiling routines (those routines are described in Section 2.3.2, and their names will appear in sans-serif font). Some events, notably `goto` and `return` statements, require more sophisticated handling.

The C language allows unrestricted jumping within a function using `goto` and `return` statements. The jump path is allowed to exit and enter arbitrary lexical blocks. To properly profile block exits and entries along a jump path, a `goto` or `return` statement is replaced by a sequence of jumps through special regions of code called *landing pads*. An *exit landing pad* is a waypoint for jumps exiting a block. It appears at the end of the block and consists of a target label and a jump (`goto`) to the exit landing pad of the nearest enclosing block. An *entry landing pad* is a waypoint for jumps entering a block. Any number of such pads may appear at the begining of a block, one for each original `goto` that enters the block. An entry landing pad consists of a target label, calls to ScopeEntry for each local variable, and a jump (`goto`) to bypass adjacent entry landing pads.

The following is a list of all program events that are instrumented, along with a description of the code inserted to handle each event:

1. Program entry/exit (in `main`)
    - Call ProfileInit/ProfileExit to initialize/clean-up the profiling library
    - Call RegisterVariable for each named variable in the program
    - Call ScopeEntry/ScopeExit for each global variable
2. Function entry/exit
    - Add exit landing pad for `return`
3. Block entry/exit
    - Call ScopeEntry/ScopeExit for each local variable[3]
    - Add entry landing pad(s) for incoming `goto`
    - Add exit landing pad(s) for outgoing `goto`/`return`

[3] `static` local variables are transformed into global variables during pre-instrumentation optimization. They are profiled as globals, not locals.

4. Read from Storage

    - Call Read after read operation

5. Write to Storage

    - Call Write after write operation

6. `goto`

    - If target is outside this block, call ScopeExit for each local variable of each nested block exited by this `goto` path
    - If target is inside a nested block, insert a `goto` to a new entry landing pad in the target block. The landing pad will contain calls to ScopeEntry for each local variable of each nested block entered by this `goto` path.
    - Landing pads will be chained to support `goto` targets in arbitrarily distant code blocks, even in non-surrounding blocks

7. `return`

    - Handled as goto to the function's exit landing pad

8. `malloc`, `calloc`, `realloc`, `free`

    - Replace by call to profileMalloc, profileCalloc, profileRealloc, or profileFree which will create/delete/update a corresponding memory image and *core* entry

9. `exit`, `abort`

    - Replace by call to ProfileEnd to emit result report

10. `setjmp`, `longjmp`

    - Replace by call to ProfileSetjmp or ProfileLongjmp to record/restore local memory images for the call chain

2.3 Run-Time Data Collection

An instrumented program performs self-profiling by calling routines from a profiling library at each relevant program event. To compute bit binding-times and other statistics, the profiling library maintains various bit-masks, flags, and counters for each variable and allocated heap objects. For each local variable, the library tracks the dynamic call chain and maintains local data structures for each scope instantiation of the variable. This section describes the run-time data structures maintained by the profiling code as well as the algorithms that use them to compute constancy statistics.

2.3.1 Data Structures

Identifying a read or write access to a given variable or heap object is complicated by the use of pointers in C. Identifying access to a variable statically, *i.e.* where it is referenced by name in the source code, is not sufficient because the same variable may also be accessed by a seemingly unrelated pointer. Similarly, a heap object may be accessed by multiple, seemingly unrelated pointers. The classic problem of pointer aliasing makes it in general impossible to identify the value of a pointer by static analysis. Hence, we must resort to dynamically identifying which variable or heap object is accessed through a dereferenced pointer.

Identifying the object reached by dereferencing a pointer is the job of the *core* data structure. The core maps an arbitrary address to the *memory image* of the object containing that address (memory images are defined below). The core is

implemented by an interval set containing the address ranges of all known objects in memory, including heap objects and all instances of local variables presently on the call stack. Objects are added to the core as they are created, namely whenever a heap object is allocated and whenever a lexical block containing local variables is entered. Objects are removed from the core as they are destroyed, namely whenever a heap object is freed and whenever a lexical block containing local variables is exited.

It is possible for a dereferenced pointer to access an object not known by the core, *e.g.* a structure created by an unprofiled library or operating system routine. The number of run-time accesses to unknown objects must be relatively small, or it will offset the statistics collected for known objects. The profiling library tallies and reports the total counts of unknown as well as known accesses.

A *memory image* embodies the profiling library's run-time information about an object in memory. The library maintains such an image for every known object, including allocated heap objects, all active instances of local variables, as well as the `argc` and `env` arrays passed to `main`. A memory image contains various bit-masks, flags, and counters used to incrementally compute bit binding-times and other statistics. Bit binding-times are represented using a collection of bit-masks that make them easy to compute incrementally using bit-parallel logic. All counters are 64-bits to prevent overflow.[4] Table 2.1 lists the components of a memory image.

A *variable image* embodies the profiling library's run-time information about a variable. Only one variable image is needed per variable, but it contains multiple memory images: one memory image to represent global, incremental results, plus

[4]For a rough calculation of how fast a profiling counter may overflow, suppose that the original, uninstrumented program runs on a 100MHz clock and issues one variable or heap access once per cycle. A 32-bit access counter would overflow after only 40 seconds of original run-time. To allow for run-times in the minutes, we must use 64-bit counters.

[5]A bit is deemed to be a sign extension if its value and the value of every more significant bit in the word have always been identical. This condition is checked whenever the bit is written.

Address Range:	*base*, *end*	— pointers
Bit Masks:	*value*	— most recent value of each bit
	defined	— true if bit was ever defined by a write
	changed	— true if bit value has changed in a write
	sign	— true if bit is a sign-extension bit in its word[5]
Counters: (per byte)	*reads*	— number of times byte was read
	writes	— number of times byte was written
Flags:	*malloced*	— true for user-allocated heap objects; not true for variables and objects allocated by the OS, *e.g.* `argc` and `argv` in `main`

Table 2.1: Components of a memory image

a stack of local memory images to represent each active instance of the variable. The global memory image is updated at each exit from the variable's scope block using that block's local memory image. A memory image actually contains only enough bit masks to represent the simplified binding-times of heap bits (Undefined, Per-Exec, Per-Def, and the signed variants thereof). A variable image contains additional bit masks that, in conjunction with the global memory image, can represent the entire bit binding-time domain. Table 2.2 lists the components of a variable image. The counters at the bottom of the table are used only for the experiment of Section 3.7 and are not present otherwise.

[6]The *enable* flag could be used for dynamically enabling and disabling the profiling of a variable at run-time. In the present implementation, it is used statically to make instrumentation simpler.

Memory Images:	*memImage*	— global, incremental memory image
	memImageStack	— stack of local memory images
Bit Masks:	*perExec*	— true for bits that have been constant throughout program execution
	perBlock	— true for bits that have been constant in each entry into their variable's block scope of definition
	perSomeBlock	— true for bits that have been constant in at least one entry into their variable's block scope and dynamic in at least one other entry
Flags:	*isConst*	— true if variable was declared `const` in the source code
	enable	— true to enable run-time profiling of variable[6]
Misc:	*name*	— variable's name
	size	— variable's size (in bytes)
Counters:	*lifetimes*	— number of times the variable's block scope was entered and the variable was instantiated
Counters:	*constLifetimes*	— number of lifetimes in which bit was constant
(per bit)	*signedLifetimes*	— number of lifetimes in which bit was a sign extension

Table 2.2: Components of a variable image. The counters in the bottom panes of the table are used only for the experiment of Section 3.7 and are not present otherwise.

Bit binding-times for variables are derived from the bit masks as follows:

$$\begin{aligned}
\textsf{Per-Exec} &= \mathit{defined} \wedge \mathit{perExec} \\
\textsf{Per-Block} &= \neg\textsf{Per-Exec} \wedge \mathit{defined} \wedge \mathit{perBlock} \\
\textsf{Sometimes-Per-Block} &= \neg\textsf{Per-Block} \wedge \mathit{defined} \wedge \mathit{perSomeBlock} \\
\textsf{Per-Def} &= \neg\textsf{Sometimes-Per-Block} \wedge \mathit{defined} \\
\textsf{Undefined} &= \neg\mathit{defined}
\end{aligned}$$

Bit binding-times for heap objects are derived from the bit masks as follows:

$$\begin{aligned}
\textsf{Per-Exec} &= \mathit{defined} \wedge \neg\mathit{changed} \\
\textsf{Per-Def} &= \mathit{defined} \wedge \mathit{changed} \\
\textsf{Undefined} &= \neg\mathit{defined}
\end{aligned}$$

A bit is a Sign variant of one of the above binding-times if and only if its *sign* bit is true.

2.3.2 Profiler Actions

Here we describe the run-time profiling routines that are called by instrumented code. The program events that are actually instrumented are listed in Section 2.2.2. The profiling routines and their actions are as follows:

ScopeEntry:

(Called on block entry for each local variable declared in the block)

- Allocate a local memory image (all bits initially undefined)
- Add memory image to *core*
- Push memory image onto variable image's stack

ScopeExit:

(Called on block exit for each local variable declared in the block)

- Pop memory image from variable image's stack
- Merge local image into global image:

$$
\begin{aligned}
read_{\text{global}} &\leftarrow read_{\text{global}} + read_{\text{local}} \\
write_{\text{global}} &\leftarrow write_{\text{global}} + write_{\text{local}} \\
value_{\text{global}} &\leftarrow value_{\text{local}} \\
defined_{\text{global}} &\leftarrow defined_{\text{global}} \vee defined_{\text{local}} \\
changed_{\text{global}} &\leftarrow changed_{\text{global}} \vee changed_{\text{local}} \\
sign_{\text{global}} &\leftarrow sign_{\text{global}} \wedge sign_{\text{local}} \\
perExec &\leftarrow perExec \wedge \neg changed_{\text{local}} \wedge (value_{\text{global}} = value_{\text{local}}) \\
perBlock &\leftarrow perBlock \wedge \neg changed_{\text{local}} \\
perSomeBlock &\leftarrow perSomeBlock \vee changed_{\text{local}}
\end{aligned}
$$

Read:

(Called after reading from a variable or dereferenced pointer)

- Get accessed object's memory image using *core*
- Increment *read* count: $reads \leftarrow reads + 1$
- Detect an unprofiled write to this object: (*e.g.* by an external library)
  if $(value_{\text{new}} \neq value)$ then call Write

Write:

(Called after writing to a variable or dereferenced pointer)

- Get accessed object's memory image using *core*

- Increment *write* count: $writes \leftarrow writes + 1$
- Mark bits as defined: $defined \leftarrow 1$
- Detect value change: $changed \leftarrow changed \vee (value_{\text{new}} \neq value)$
- Remember new value: $value \leftarrow value_{\text{new}}$

profileMalloc, profileCalloc:

(Called instead of `malloc`, `calloc`)

- Allocate heap object (call `malloc`/`calloc`)
- Create+initialize new memory image and insert into *core*

profileRealloc:

(Called instead of `realloc`)

- Realocate heap object (call `realloc`)
- Update corresponding memory image in *core*

profileFree:

(Called instead of `free`)

- Tally total reads, writes, *etc.*
- Deallocate heap object (call `free`)
- Remove corresponding memory image from *core*

profileSetjmp:

(Called instead of `setjmp`)

In UNIX, `setjmp` is used to save the present call stack, and *longjmp* is used to jump back to that point in the program from any point deeper in the call stack. To handle these jumps, the profiler must save and restore the active local memory images of each variable. Because local memory images

are already kept on a stack in each variable image, it suffices to save only the position on each stack.

To create a jump environment:

- For each variable, record the stack pointer from its stack of local memory images
- Call conventional `setjmp`

profileLongjmp:

(Called instead of `setjmp`)

To restore a jump environment:

- For each variable, pop the stack of local memory images until restoring the stack pointer from the jump environment
- Call conventional `longjmp`

profileEnd:

(Called before `exit`, `abort`, and before falling off the end of `main`).

- For each variable, repeatedly pop the stack of local memory images and call ScopeExit on the popped image (this simulates a clean exit even if the exit is from a deeply nested code block)
- For each heap object, call profileFree to deallocate it
- Emit result report (see Section 2.4)

2.3.3 Cost of Profiling

The profiling approach described herein has a nontrivial cost in memory usage and run-time. Instrumentation for self-profiling expands the code size several-fold due to the addition of profiler function calls. Data size increases dramatically

due to the overhead of memory and variable images. Assuming minimum size representations for the images of Tables 2.1 and 2.2, an N-bit heap object would require $13 + 16N$ bytes, and an N-bit variable would require at least $35 + 19N$ bytes.[7] Thus, a typical 32-bit variable would require at least 111 bytes of image— an overhead factor of about 28. The present implementation is significantly worse due to indirection in the image representation and due to memory alignment. In the implementation, a typical 32-bit variable requires at least 280 bytes— an overhead factor of 70.

Empirically, the total run-time memory requirements of an instrumented program are seen to be 100 to 300 times larger than that of the original, uninstrumented program. The actual run-time for an instrumented program is seen to be approximately 1000 times larger than that of the original. Run-time may be larger still if physical memory is insufficient and the program swaps to disk under virtual memory. Experiments for this project were run on a machine with 2 gigabytes of physical memory, allowing for swap-free profiling of programs which originally require no more than about 20 megabytes.

2.4 Run-Time Result Report

Before exiting, an instrumented program invokes the profileEnd profiling routine to emit a result report to disk. This textual report summarizes all access counts and bit binding-time information collected during execution. Parts of the report may be imported directly into a spreadsheet for graphing (this is in fact how the graphs of Chapter 3 (Results) are generated). The result report includes the following information:

[7]The minimum size of a variable image is $35 + 19N$ bytes assuming an empty name string, no lifetime counters, and no recursion. Each dynamic level of recursion adds a local memory image of $13 + 16N$ additional bytes.

1. Unknown accesses:

    - Read and write counts for unknown objects

2. Heap statistics:

    - Breakdown of bits by bit binding-time
    - Breakdown of bit-reads by bit binding-time

    These counts reflect all allocations and access on the heap throughout execution. Bit binding-times are from the restricted heap domain (Undefined, Per-Exec, Per-Def).

3. Variable statistics:

    - Breakdown of bits by bit binding-time.
    - Breakdown of bit-reads by bit binding-time.
    - Breakdown of bit-reads to constant bit regions by bit position.

      This only considers Per-Exec and Per-Exec, Sign bits. The region positions reported are high-order, low-order, and entire word, with all remaining bit-reads binned as "elsewhere".[8]
    - Breakdown of variables by read count.

      This is a cumulative distribution of word-reads, binned and sorted by the read count of each variable. It is intended to verify the 90-10 rule that most word-reads are to a few, frequently-read variables.
    - Breakdown of variable bits by read count.

      This is a cumulative distribution of bit-reads, binned and sorted by the read count of each bit's variable. It is different from the previous bullet

[8]The breakdown of bit-reads by bit region position replaces an earlier analysis that emitted a 2D histogram of bit-reads binned by the LSB and MSB positions of each bit's surrounding bit region.

because variables have different bit widths. It is intended to verify the 90-10 rule that most bit-reads are to a few, frequently-read bits.

- Complete variable dump.

  This lists critical statistics for each variable: bit width, read/write counts, lifetime count, ratios of these counts, a binding-time bitmap, and the variable's fractional contribution to bit-reads in each binding-time. Variables are sorted by read count.

2.5 Call Chain Disambiguation

The approach of merging bit constancy information from all run-time instances of a variable is conservative. It ignores the possibility that the constancy behavior of a variable may be vastly different depending on how its lexical block is reached in the dynamic call chain. For instance, a function `f` may be called with constant arguments from function `A` but with dynamic arguments from function `B`. The `A` context of the function `f` could be specialized, but the `B` context should not be.

This section discusses a program transformation that adds context sensitivity to the profiling analysis. The transformation is applied before instrumentation, and the target program proceeds normally through the profiling methodology. The transformation disambiguates the call chain leading to any given C function by replicating and binding the function separately for each possible call path. In effect, this turns the static call graph into a tree (in the abscence of recursion or reentrance). Local variables are thus disambiguated and will be analyzed independently for each path in the call chain that uses them.

No attempt is made to merge bit constancy information for corresponding, disambiguated variables. The experiment is only concerned with whether this disambiguating transformation significantly affects the bit binding-time profile

of the entire program. The assumption is that a unique call path, if executed frequently enough to warrant specialization, would significantly affect the total binding-time profile of the program.

Call chain disambiguation may allow a compiler to perform better inter-procedural optimization. Some disambiguated call paths could be folded away in their entirety by an optimizing compiler. If this were done by SUIF's pre-instrumentation optimizations, it would simply remove certain functions and variables from the profiled program.[9] Such folding may also be done by the binary compiler in post-instrumentation optimization. Thus, although procedural replication may increase code size, it is ultimately seen to reduce the run-time of most profiled programs.

[9]The version of SUIF and extensions used in this project implementation do not, to the author's knowledge, perform inter-prcedural optimizations.

Chapter 3

Results

3.1 Profiled Applications

The applications profiled in this project are primarily multimedia processing tasks, namely signal processing, compression, and encryption. Such applications tend to be data-path intensive, *i.e.* they spend most of their time processing large data streams in a small set of computational kernels. This program form can benefit significantly from specialization, since any savings realized in a specialized kernel are amplified by reuse of that kernel.

We hypothesize that typical media processing tasks exhibit particular behaviors that are computationally wasteful at the bit level. Many of these programs implement communication and storage standards and must deal with quantities of irregular bit width. When implemented for a word-based architecture such as a microprocessor, these programs must use the next largest available data width, leaving many high-order bits unused. For instance, audio tasks on 16-bit samples that need one or two overflow bits must resort to 32-bit computation. Encryption tasks face a similar problem whenever custom bit widths or direct bit manipulation is used. Furthermore, processing media streams typically involves sequential

access to blocks of memory, where array indexes or pointers are range-limited (zero high-order bits) and/or strided (constant low-order bits).

Initial exploration with the profiling system was done with GNU's `gzip` 1.2.4 [21] (an LZW-based compressor/decompressor) and Berkeley's `mpegencode` suite [17] (a video compressor for the MPEG-1 standard). The bulk of experimentation was done with UCLA's *MediaBench* suite [29], a collection of 11 multimedia processing benchmarks including signal processing, encryption, and image manipulation tasks. The MediaBench components which were profiled include:

ADPCM `rawcaudio`, `rawdaudio`

Adaptive differential pulse code modulation for audio coding (compresses 16-bit samples into 4-bit code words)

EPIC `epic`, `unepic`

Lossy, wavelet-based image compression

G721 `encode`, `decode`

Voice compression for CCITT G.711, G.721, and G.723 standards

GSM `toast`, `untoast`

Speech transcoding as per the European GSM 06.10 standard (compresses frames of 160 13-bit samples into 260 bit code words)

JPEG `cjpeg`, `djpeg`

Lossy, DCT-based image compression

MPEG `mpeg2encode`, `mpeg2decode`

Lossy video compression as per the MPEG-2 standard

Several MediaBench components were not profiled, due to technical problems discussed in section 4.1.1. They include:

Ghostscript	PostScript interpreter
Mesa	3D library implementing the OpenGL standard, with 3 applications (*mipmap*, *osdemo*, *texgen*)
Pegwit	Public-key encryption using elliptic curves, SHA1, and symmetric block cypher
PGP	Public-key encryption using IDEA, MD5, and RSA
RASTA	Speech recognition using PLP, RASTA, and Jah-RASTA techniques

To study more general purpose applications, we attempted to profile the SPECint95 suite [40]. However, most of those programs, could not successfully pass through the profiling system, for reasons discussed in section 4.1.1. No SPECint95 findings are presented here.

3.2 Ninety-Ten Rule for Bit Storage

Specialization is a relatively difficult task, requiring non-trivial time and memory resources for recompilation. It is probably impractical to specialize all parts of a program. Rather, we desire situations where specializing a small part of a program (its kernels) will yield large performance gains. Whereas traditional profiling identifies kernel code blocks, storage-based profiling instead identifies *kernel variables*. Kernel variables are the most frequently accessed storage elements of a program. Presumably, they serve as input and/or output to kernel code, but may also be accessed elsewhere.

The classic "90-10 rule" claims that a program typically spends 90% of its time executing 10% of its code. We wish to verify a corresponding rule for storage. Does a program typically spend 90% of its reads accessing the same 10% of its

variables? Similarly, at the bit level, does a program typically spend 90% of its bit-reads accessing the same 10% of its variable bits? These two questions should be distinguished because different variables have different bit widths (this discrepancy is especially evident among aggregates and arrays).

Figures 3.1 and 3.2 attempt to verify the "90-10 rule" for variables and variable bits, respectively, in MediaBench. Figure 3.1 is a cumulative distribution of variable reads, sorted by the read count of each variable. Essentially, it graphs what percentage of variables are responsible for what percentage of variable read operations. Variables are sorted along the horizontal access in descending order of read count, so the most frequently-read variable are tallied first. The thick curve represents a mean average over all profiled applications. We find that, on average, 90% of variable read operations access the same 12% of variables.

Figure 3.2 is the bit-level analogue of Figure 3.1. It is a cumulative distribution of bit-reads to variables, sorted by the read count of each bit's variable. We find that, on average, 90% of all bit-read operations on variables access the same 16% of bits. In both figures, these findings are in line with the proverbial "90-10 rule."

The only rule breaker in Figure 3.1 is ADPCM (audio coding), where 90% of variable reads access 28% of variables. The dilution of the "90-10 rule" arises because MediaBench's ADPCM implementation is essentially nothing but kernels. ADPCM is the smallest MediaBench application, having only 3 routines (`main`, encode, and decode) and 36 profiled variables. Hence, each kernel accesses a large fraction of the program's variables, and no single variable is most important.

The rule breakers in Figure 3.2 include JPEG (encode and decode) as well as EPIC and MPEG-2 decoding, in which 90% of bit reads access 20%-30% of variable bits. Here, the dilution of the "90-10 rule" is apparently to the use of tables, implemented as statically declared arrays. Table accesses, which comprise a large fraction of each program's bit-reads, are spread out over the many bits

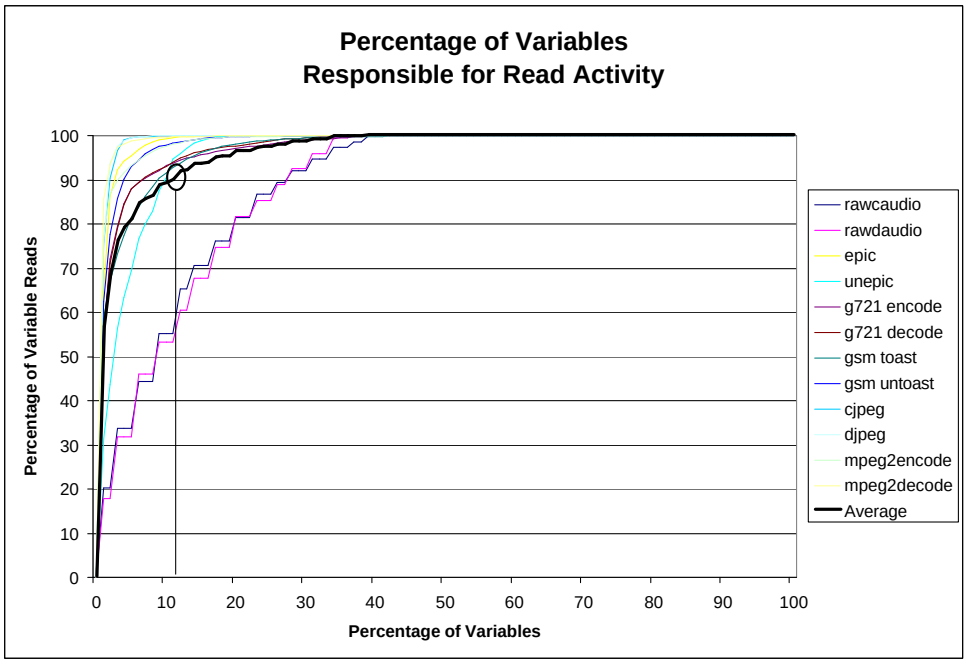


Figure 3.1: Ninety-Ten Rule: Cumulative distribution of variables responsible for word reads

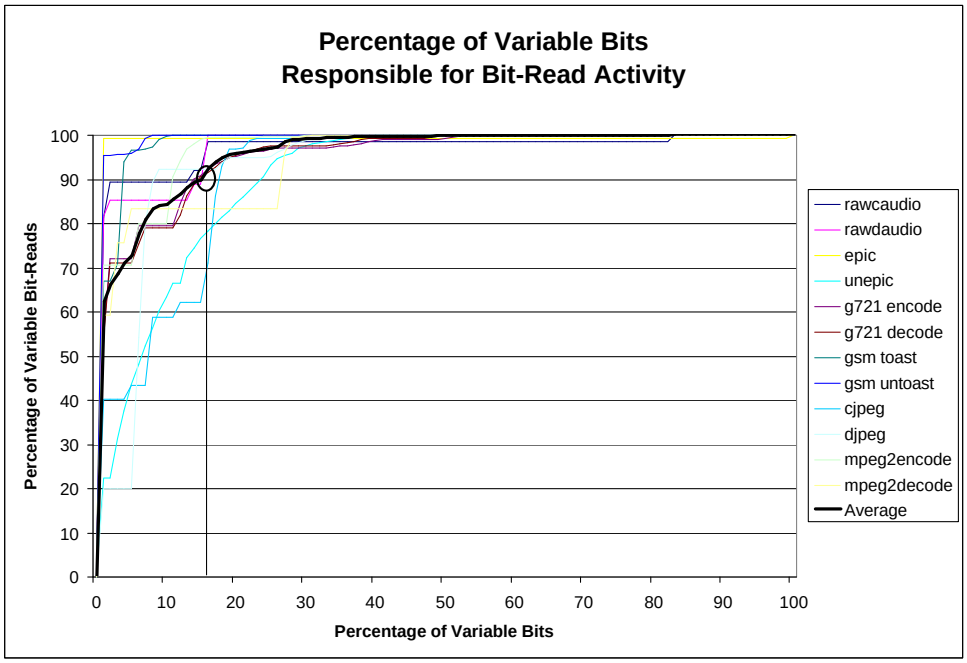


Figure 3.2: Ninety-Ten Rule: Cumulative distribution of variable bits responsible for bit-reads

of the tables. No particular table or table region is solely responsible for an overwhelming majority of bit-reads.

One should note that even the rule breakers in these figures exhibit great reuse of storage. In ADPCM, 50% of all variable read operations access the same 9% of variables. Similarly, in EPIC, JPEG, and MPEG-2, 50% of all variable bit-reads access no more than 7% of variable bits (in the case of MPEG-2 decode, that number is less than 1% of variable bits).

3.3 Bit Constancy on the Heap

The analysis performed for variables is different than the analysis performed for the heap. The analyses involve different bit binding-time domains and somewhat different profiling actions. Although experiments were run for both variables and heap storage, most of the results in this chapter focus on variables. The reason for this bias is twofold. First, we observe that among the profiled applications, reads to variables occur far more frequently than reads to the heap. Furthermore, we observe that bit constancy in variables is fairly uniform across applications, whereas bit constancy on the heap is highly application dependent. It appears that bit-level computational waste in variables is endemic to the MediaBench benchmark suite, possibly to a larger class of programs written in high-level languages. Hence, this waste in variables is more interesting than waste on the heap, as a language phenomenon and for its specialization potential.

Table 3.1 presents the breakdown of bit-read operations in MediaBench among variables, heap objects, and unknown objects. We find that, on average, a program spends some 95% of its bit-reads accessing variables and only 5% accessing heap objects. Some applications do not use heap storage at all, and no application spends more than 12% of its bit-reads on the heap. Thus variable reads are

significantly more frequent than heap reads, on average 20 times more frequent. Table 3.1 also demonstrates that read accesses to unknown (unprofiled) objects are practically non-existent. Unprofiled objects would include any objects not visible at compile-time, *e.g.* structures returned by library or operating system routines.

Table 3.2 summarizes the bit constancy behavior observed on the heap in MediaBench applications. We consider only two binding-times: bits whose value is constant (Per-Exec) and bits whose value is dynamic (Per-Def) during the allocation lifetime of the enclosing heap object. We find that the fraction of constant bits on the heap is highly application dependent. Similarly, the fraction of bit-reads to constant bits on the heap is highly application dependent. Some applications maintain and reuse a lot of constant heap storage, for instance JPEG compression (`cjpeg`), where 94.2% of heap bits and 88.5% of heap bit-reads are constant valued. This constant storage might consist of the input image, quantization tables, and Huffman tables. Other applications use mostly dynamic heap storage, for instance GSM speech transcoding (`encode`) where 98.5% of heap bits and 99.95% of heap bit-reads are dynamic valued.

Bit constancy on the heap may be difficult to exploit by specialization because of the sheer size of objects on the heap. While it may be practical to specialize away a small lookup table (ROM) by hard-wiring its values in logic, it is not at all practical for large tables with many values. When creating custom hardware, the choice between a ROM and logic implementation for a lookup table involves a time-space tradeoff which is beyond the scope of this paper.

	Total Reads (Millions)	Variable Reads	Heap Reads	Unknown Reads
rawcaudio	177	100.0%	0.0%	0.0%
rawdaudio	133	100.0%	0.0%	0.0%
epic	6590	89.3%	10.7%	*
unepic	625	88.0%	12.0%	0.0%
g721 encode	4860	100.0%	0.0%	0.0%
g721 decode	4590	100.0%	0.0%	0.0%
gsm toast	2170	89.8%	10.2%	*
gsm untoast	893	97.5%	2.5%	*
cjpeg	302	91.2%	8.8%	*
djpeg	78	89.2%	10.8%	*
mpeg2encode	25600	91.6%	8.4%	*
mpeg2decode	3620	99.6%	0.4%	0.0%
Average	4140	94.7%	5.3%	0.0%

Table 3.1: Breakdown of bit-read operations among variables, heap, and unknown objects (* indicates a non-zero quantity less than 0.01%)

	Heap Bits (Millions)	Const Bits	Dynamic Bits	Heap Reads (Millions)	Const Reads	Dynamic Reads
rawcaudio	–	–	–	–	–	–
rawdaudio	–	–	–	–	–	–
epic	24.5	79.7%	20.3%	708	37.1%	62.9%
unepic	23.7	82.7%	17.3%	75.0	70.8%	29.2%
g721 encode	–	–	–	–	–	–
g721 decode	–	–	–	–	–	–
gsm toast	0.005	1.5%	98.5%	221	0.05%	99.95%
gsm untoast	0.003	2.1%	97.9%	22.2	0.5%	99.5%
cjpeg	1.1	94.2%	5.8%	26.6	88.5%	11.5%
djpeg	0.2	67.1%	32.9%	8.4	77.6%	22.4%
mpeg2encode	10.0	64.9%	35.1%	2150	84.5%	15.5%
mpeg2decode	3.1	77.3%	22.7%	14.5	75.1%	24.9%

Table 3.2: Bit binding-times on the heap (– indicates zero heap use)

3.4 Bit Constancy in Variables

We now summarize the bit binding-times found among variables in the MediaBench programs. Table 3.3 lists the breakdown of variable bits by binding-time. It shows what fraction of storage falls into each binding-time. Table 3.4 lists the breakdown of variable bit-reads by binding-time. It shows what fraction of dynamic bit-reads are made to bits of each binding-time. The tables also total the contribution of all sign bits and Per-Exec bits, since these are the bits that should be easiest to specialize. The corresponding Figures 3.3 and 3.4 graph the contribution of bits and bit-reads from each binding-time cumulatively. The bit binding-times are sorted on the horizontal axis by frequency of change, as per the partial ordering of the binding-time domain $\mathcal{B}$. The heuristic information in these graphs is that convex curves indicate abundant constancy, whereas concave curves represent more dynamic behavior. Note, the tables and figures do not tally any allocated storage that remains undefined and is never read.

We find significant contribution from Per-Exec bits that never change value during execution. Table 3.3 shows that, on average, 66.4% of all variable bits are bound Per-Exec (28.1% being sign extension bits, 38.3% being non-sign). Table 3.4 shows that these bits account for an average 50.4% of dynamic bit-reads to variables (40.5% reading sign-extension bits, 10.1% reading non-sign). Thus, across all the profiled applications, typically more than half of dynamic bit-reads are re-reading bits that never change.

We also find a significant contribution from sign-extension bits. Table 3.3 shows that 33% on average and as many as 99.6% of variable bits are sign bits. Table 3.4 shows that these bits account for an average 56.1% of dynamic bit-reads to variables. This is a promising prospect for specialization, since an arithmetic data-path needs only one representative sign bit. A single sign bit is $1/n$ of the bits in an n-bit word (*e.g.* $1/8 = 12.5\%$ in a byte, $1/64 = 1.6\%$ in a 64-bit word).

	Variable Bits (Thousands)	const	Per Exec, signed	Per Exec	Per Block, Signed	Per Block	Sometimes Per Block, Signed	Sometimes Per Block	Per Def, Signed	Per Def	Sign	Per Exec or Sign
rawcaudio	23.9	0.0%	1.7%	26.3%	0.0%	0.0%	0.2%	0.2%	8.1%	63.5%	10.0%	36.3%
rawdaudio	23.9	0.0%	1.7%	13.7%	0.0%	0.0%	0.2%	0.2%	8.1%	76.1%	10.0%	23.7%
epic	2110.0	0.0%	99.6%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	99.6%	99.9%
unepic	3.5	0.0%	41.7%	38.1%	0.0%	4.5%	0.1%	1.3%	2.0%	12.3%	43.7%	81.8%
g721 encode	7.0	0.0%	16.0%	55.5%	4.0%	17.6%	0.3%	1.3%	0.9%	4.5%	21.2%	76.7%
g721 decode	6.6	0.0%	14.5%	57.9%	3.2%	17.3%	0.3%	1.1%	0.9%	4.7%	19.1%	76.9%
gsm toast	130.0	0.0%	3.5%	88.3%	0.5%	1.4%	0.7%	3.0%	0.3%	2.2%	5.1%	93.3%
gsm untoast	125.0	0.0%	3.1%	90.7%	0.7%	1.9%	0.3%	0.6%	0.3%	2.4%	4.4%	95.1%
cjpeg	61.7	27.0%	36.4%	15.7%	0.0%	1.8%	2.1%	2.0%	12.2%	2.9%	50.7%	66.3%
djpeg	41.2	21.8%	34.3%	25.8%	3.8%	9.2%	0.3%	2.1%	0.4%	2.3%	38.8%	64.6%
mpeg2encode	84.0	0.0%	38.0%	30.6%	0.4%	7.0%	1.0%	2.1%	8.1%	12.8%	47.6%	78.2%
mpeg2decode	172.0	0.0%	46.2%	16.2%	0.1%	2.6%	0.2%	0.3%	5.5%	28.8%	52.1%	68.4%
Average	232.0	4.1%	28.1%	38.3%	1.1%	5.3%	0.5%	1.2%	3.9%	17.7%	33.5%	71.8%

Table 3.3: Breakdown of variable bits by bit binding-time

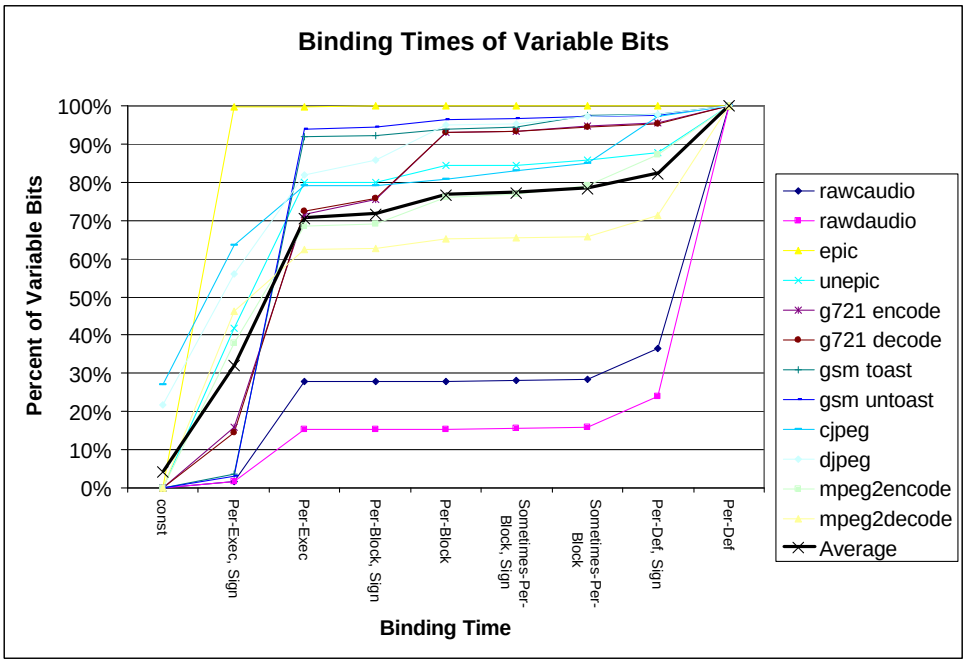


Figure 3.3: Cumulative distribution of variable bits by bit binding-time

	Variable Bit-Reads (Millions)	const	Per Exec, signed	Per Exec	Per Block, Signed	Per Block	Sometimes Per Block, Signed	Sometimes Per Block	Per Def, Signed	Per Def	Sign	Per Exec or Sign
rawcaudio	177	0.0%	37.3%	6.6%	0.0%	0.0%	15.4%	14.4%	7.7%	18.6%	60.4%	67.0%
rawdaudio	133	0.0%	44.6%	8.3%	0.0%	0.0%	18.6%	11.4%	2.8%	14.3%	66.0%	74.3%
epic	5880	0.0%	51.0%	6.5%	0.0%	7.5%	0.1%	11.2%	3.1%	20.7%	54.1%	60.6%
unepic	550	0.0%	36.0%	18.0%	0.0%	1.4%	0.0%	2.9%	4.7%	37.0%	40.7%	58.7%
g721 encode	4860	0.0%	48.9%	19.5%	11.3%	9.4%	0.4%	6.3%	0.4%	3.8%	61.0%	80.4%
g721 decode	4590	0.0%	55.5%	19.8%	4.6%	9.0%	0.4%	6.2%	0.4%	3.9%	61.0%	80.8%
gsm toast	1950	0.0%	33.8%	3.6%	0.5%	0.8%	14.7%	32.2%	2.4%	12.1%	51.3%	54.9%
gsm untoast	871	0.0%	13.9%	12.5%	0.1%	0.5%	16.6%	15.8%	22.7%	17.8%	53.3%	65.9%
cjpeg	276	3.4%	45.9%	7.6%	0.2%	3.1%	14.3%	11.2%	2.6%	11.7%	63.0%	70.6%
djpeg	69	0.8%	37.6%	9.5%	3.1%	2.7%	6.4%	25.1%	2.0%	12.8%	49.1%	58.5%
mpeg2encode	23500	0.0%	33.8%	2.9%	0.1%	3.3%	29.6%	25.6%	0.1%	4.7%	63.6%	66.4%
mpeg2decode	3610	0.0%	47.2%	6.9%	0.2%	7.9%	1.4%	17.5%	1.2%	17.7%	50.0%	56.9%
Average	3870	0.3%	40.5%	10.1%	1.7%	3.8%	9.8%	15.0%	4.2%	14.6%	56.1%	66.3%

Table 3.4: Breakdown of variable bit-reads by bit binding-time

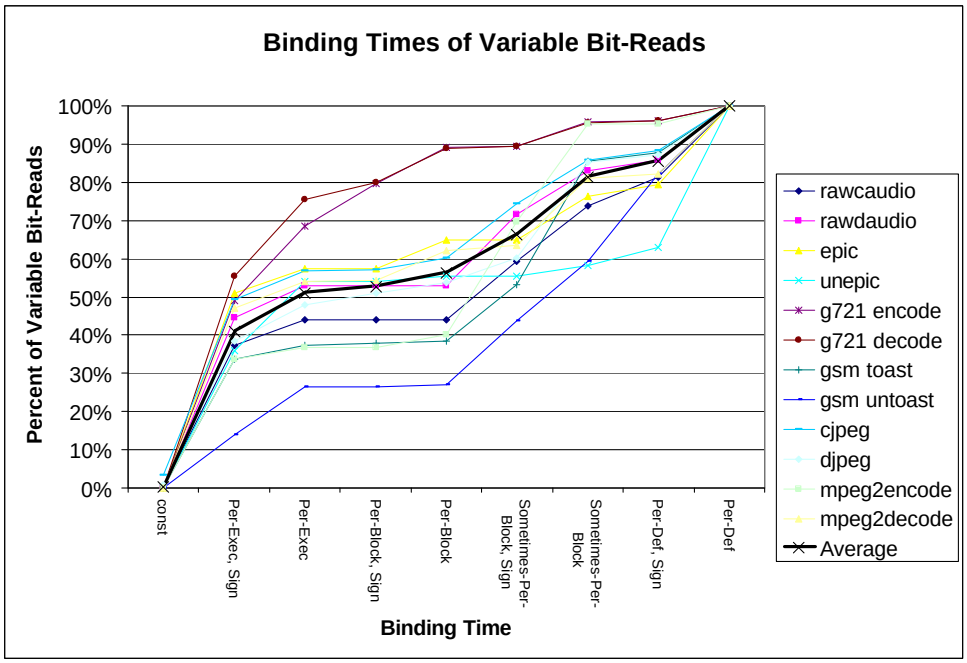


Figure 3.4: Cumulative distribution of variable bit-reads by bit binding-time

Hence the fact that 33% of all variable bits are sign bits indicates that there are many more sign bits than necessary in each word.

Altogether, we find that some 70% of dynamic bit-reads in variables are to easily-identified constant and sign data. On average, 56.1% of bit-reads are to sign-extension bits that could be specialized by narrower data-paths. An additional 10.1% of bit-reads are to Per-Exec bits that could be specialized once per execution (possibly once across all executions, since we do not know how many of these bits are actually Per-Compile). An additional 3.8% of bit-reads are to Per-Block bits that could be specialized once per entry into their variable's scope (*e.g.* once per function entry, once per loop iteration, *etc.*). Together, these are 70%.

The remaining 30% of bit-reads in variables are to dynamic quantities that are less promising for specialization. They are split fairly evenly between Per-Def and Sometimes-Per-Def bits. The more slowly-changing bits in these categories may still be amenable to probabilistic specialization techniques, as described in section 4.4.3.

It is interesting to note that nearly none of the MediaBench applications use `const` variables. The only exception is the JPEG suite (cjpeg, djpeg) which uses numerous `const` arrays, *e.g.* for tables of DCT coefficients. The typical under-use of explicitly declared constants may be due to the fact that C has no syntax to declare dynamically-valued constants (DVCs). Schilling [43] suggests that this under-use is due to bad programming practice, since it is common even in languages like C++ which allow declaring DVCs. If MediaBench indeed exhibits Schilling's problem of having many candidate but undeclared DVCs, then we should expect to find a large contribution from DVC words where all bits are either Per-Block or Per-Exec (possibly signed). This question is partially answered in Section 3.5, where Table 3.5 shows that only 7% of Per-Exec bit-reads are to fully Per-Exec words. However, we have not tallied bit-reads to words that

also allow Per-Block bits. If the fraction of bit-reads to such DVC words were large, then this would be an argument for the addition of DVC declaration as a language feature. Note that Schilling reports that 40% of local variables were DVC candidates, but he does not report how often they were read, so there is no basis of comparison with our measure of bit-reads.

3.5 Constancy in Bit Regions of Variables

The bit position of unchanging or slowly varying bits affects their potential impact for specialization. Of particular interest are sign extension bits, since they may allow many high-order bits to be removed from a data-path. More generally, we are interested in contiguous regions of unchanging bits inside a word, because specializing around them may significantly speed up arithmetic carry chains and other cascades. Unchanging bit regions in the most significant position are typical of range-limited arithmetic (they may be sign or constant). Unchanging bit regions in the least significant position are typical of arithmetic involving multiples of 2^n, for instance in pointer arithmetic.

Table 3.5 and the corresponding Figure 3.5 show bit-reads to constant bit regions of variables in MediaBench. They show how many bit-reads were actually to bits in contiguous regions of Per-Exec and/or Per-Exec, Sign bits. We differentiate between high-order bit regions (that include MSB), low-order bit regions (that include LSB), and entire-word bit regions (that include all bits of a word). Bit-reads to Per-Exec bits in other positions are tallied in the "elsewhere" column. The total contribution of bit-reads in Per-Exec and/or Per-Exec, Sign bits is copied from Table 3.4 for reference.

We find that bit-reads to bits defined once per execution are due primarily to constant high-order bit regions. Of the average total 50.6% of bit-reads from

	Low Order Bit Region	High Order Bit Region	Entire Word	Else-where	Total	Per Exec, Signed	Per Exec	Non-sign High Order
rawcaudio	0.6%	37.7%	5.3%	0.3%	43.9%	37.3%	6.6%	5.7%
rawdaudio	0.6%	45.2%	7.1%	0.0%	52.9%	44.6%	8.3%	7.8%
epic	3.6%	48.1%	3.9%	1.9%	57.5%	51.0%	6.5%	1.0%
unepic	7.8%	33.2%	12.5%	0.5%	54.0%	36.0%	18.0%	9.7%
g721 encode	1.9%	55.4%	10.5%	0.6%	68.4%	48.9%	19.5%	17.0%
g721 decode	0.8%	52.1%	21.8%	0.6%	75.4%	55.5%	19.8%	18.4%
gsm toast	0.3%	36.2%	0.7%	0.2%	37.3%	33.8%	3.6%	3.1%
gsm untoast	0.3%	16.4%	9.5%	0.2%	26.4%	13.9%	12.5%	12.0%
cjpeg	1.1%	40.0%	9.3%	3.1%	53.5%	45.9%	7.6%	3.4%
djpeg	0.7%	41.5%	3.8%	1.1%	47.1%	37.6%	9.5%	7.7%
mpeg2encode	1.3%	33.8%	0.2%	1.4%	36.7%	33.8%	2.9%	0.2%
mpeg2decode	0.3%	47.1%	0.2%	6.6%	54.1%	47.2%	6.9%	0.0%
Average	1.6%	40.6%	7.1%	1.4%	50.6%	40.5%	10.1%	7.2%

Table 3.5: Variable bit-reads to constant, contiguous bit regions containing Per-Exec and Per-Exec, Sign bits. Percentages are taken as fractions of all variable bit-reads in the application.

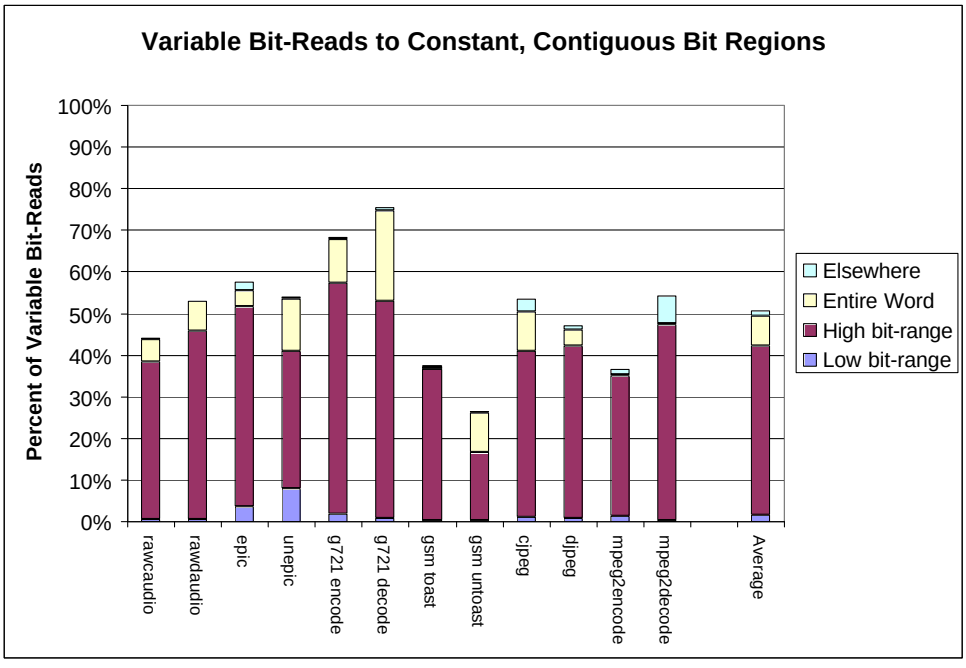


Figure 3.5: Variable bit-reads to constant, contiguous bit regions containing Per-Exec and Per-Exec, Sign bits. Percentages are taken as fractions of all bit-reads in the application.

Per-Exec and Per-Exec, Sign bits, 40.6% are to bits in high-order bit regions. Only 1.6% are to constant low-order bit regions, and 1.4% are to elsewhere in a word. A modest 7.1% of bit-reads are to entirely constant words.

Are bits in high-order regions really all sign bits? Or are there also constant, non-sign bits in high-order regions? These non-sign bits would be in the less significant positions of high-order regions, flanked to the left by sign bits. Unlike the sign bits, constant non-sign bits cannot simply be removed from a data-path. They must be specialized by other means such as bit-level constant folding. It is possible to find the contribution of constant, non-sign, high-order bit-reads from simple arithmetic on table columns. Sign bits must come from high-order and entire-word regions, so we calculate non-sign, high-order bit-reads as:

$$(\text{Non-Sign High Order}) = (\text{High Order Bit Region}) + (\text{Entire Word}) - (\text{Per-Exec, Sign})$$

The result, shown in the rightmost column, indicates that an average 7.2% of bit-reads are to Per-Exec (non-sign) bits in high-order bit regions. This is only about 1/6 of the 40.6% of bit-reads in constant bit regions. The remaining 5/6 are to sign bits.

3.6 Effects of Call Chain Disambiguation

Tables 3.6 and 3.7 show bit binding-times among variables in MediaBench when using call chain disambiguation (as described in Section 2.5). The tables are analogous to Tables 3.3 and 3.4 which tally variable bits and bit-reads without call chain disambiguation. Similarly, the corresponding Figures 3.6 and 3.7 are analogous to Figures 3.3 and 3.4. There is no data collected for application `mpeg2decode`, since call chain disambiguation could not be applied to it successfully.

Table 3.6 shows that call chain disambiguation has negligible effect on the size

	Variable Bits (Thousands)	const	Per Exec, signed	Per Exec	Per Block, Signed	Per Block	Sometimes Per Block, Signed	Sometimes Per Block	Per Def, Signed	Per Def	Sign	Per Exec or Sign
rawcaudio	23.9	0.0%	1.7%	26.3%	0.0%	0.0%	0.3%	0.2%	8.1%	63.6%	10.0%	36.2%
rawdaudio	23.9	0.0%	1.7%	13.7%	0.0%	0.0%	0.2%	0.2%	8.1%	76.1%	4.0%	23.7%
epic	2110.9	0.0%	99.6%	0.3%	0.0%	0.02%	0.0%	0.01%	0.01%	0.09%	99.6%	99.9%
unepic	3.5	0.0%	43.7%	37.3%	0.0%	3.7%	0.06%	1.3%	1.9%	12.0%	45.7%	83.0%
g721 encode	6.1	0.0%	12.6%	63.1%	3.2%	15.0%	0.0%	0.0%	1.0%	5.2%	16.8%	79.9%
g721 decode	5.9	0.0%	11.7%	64.9%	2.5%	14.6%	0.0%	0.0%	1.1%	5.3%	15.3%	80.2%
gsm toast	127	0.0%	2.8%	90.2%	0.5%	1.4%	0.5%	2.2%	0.4%	2.2%	4.1%	94.3%
gsm untoast	124	0.0%	3.3%	91.2%	0.7%	1.8%	0.2%	0.4%	0.2%	2.2%	4.4%	95.6%
cjpeg	59.5	28.9%	37.1%	12.4%	0.1%	1.9%	2.2%	2.0%	12.6%	2.8%	52.0%	64.4%
djpeg	42.0	25.3%	32.3%	25.0%	3.8%	9.3%	0.3%	1.7%	0.3%	2.0%	36.7%	61.7%
mpeg2encode	77.9	0.0%	36.4%	34.3%	0.5%	7.3%	0.3%	0.3%	8.7%	12.2%	45.9%	80.1%
mpeg2decode	–	–	–	–	–	–	–	–	–	–	–	–
Average	236.0	4.9%	25.7%	41.7%	1.0%	5.0%	0.4%	0.8%	3.9%	16.7%	30.9%	72.6%

Table 3.6: Call chain disambiguation: Breakdown of variable bits by bit binding-time

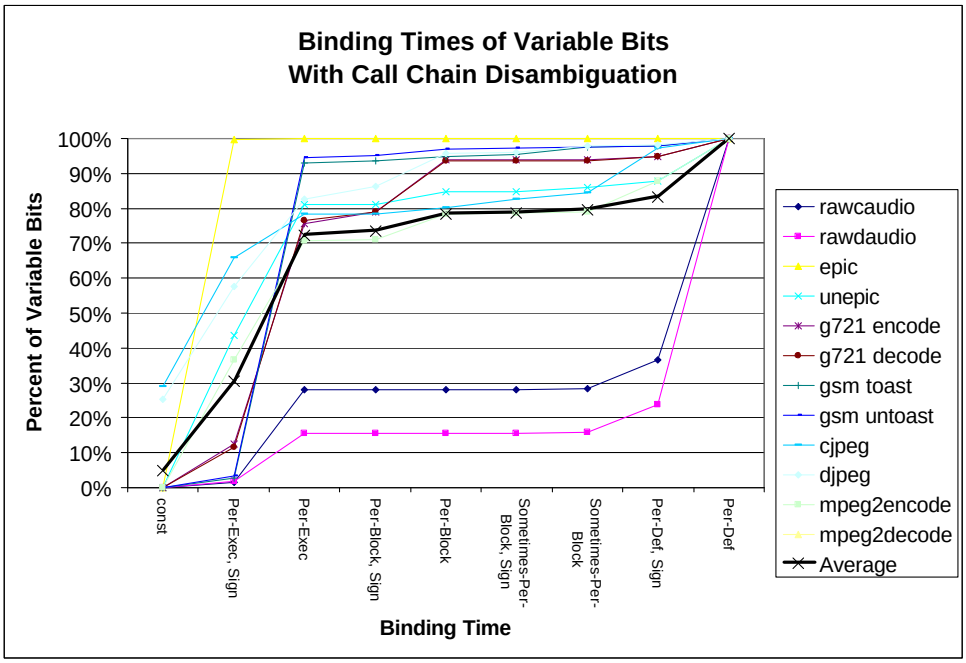


Figure 3.6: Call chain disambiguation: Cumulative distribution of variable bits by bit binding-time

	Variable Bit-Reads (Millions)	const	Per Exec, signed	Per Exec	Per Block, Signed	Per Block	Sometimes Per Block, Signed	Sometimes Per Block	Per Def, Signed	Per Def	Sign	Per Exec or Sign
rawcaudio	177	0.0%	37.3%	5.7%	0.0%	0.0%	15.4%	14.4%	7.7%	19.6%	60.4%	66.1%
rawdaudio	133	0.0%	44.6%	7.2%	0.0%	0.0%	18.6%	12.5%	2.8%	14.3%	66.0%	73.2%
epic	1450	0.0%	42.2%	12.4%	0.03%	7.3%	1.5%	22.6%	0.1%	13.9%	43.8%	56.2%
unepic	550	0.0%	36.6%	17.7%	0.0%	1.3%	0.0%	2.9%	4.7%	36.8%	41.4%	59.0%
g721 encode	498	0.0%	17.8%	52.3%	0.0%	0.8%	0.0%	0.0%	3.8%	25.2%	21.7%	73.9%
g721 decode	474	0.0%	17.6%	51.4%	0.0%	0.8%	0.0%	0.0%	4.0%	26.2%	21.5%	73.0%
gsm toast	138	0.0%	3.2%	0.6%	1.6%	3.4%	10.0%	61.2%	5.1%	15.0%	19.9%	20.4%
gsm untoast	30.1	0.0%	6.0%	1.6%	2.4%	6.7%	7.1%	63.3%	0.8%	12.1%	16.3%	17.9%
cjpeg	253	3.7%	45.5%	6.2%	0.0%	2.9%	15.6%	10.9%	2.8%	12.5%	63.8%	70.0%
djpeg	66.4	0.8%	37.0%	8.7%	3.3%	2.8%	6.6%	25.1%	2.1%	13.7%	49.0%	57.7%
mpeg2encode	959	0.0%	8.9%	25.5%	0.5%	28.4%	0.01%	0.09%	0.9%	35.8%	10.2%	35.7%
mpeg2decode	–	–	–	–	–	–	–	–	–	–	–	–
Average	430	0.4%	27.0%	17.2%	0.7%	4.9%	6.8%	19.4%	3.2%	20.5%	37.6%	54.8%

Table 3.7: Call chain disambiguation: Breakdown of variable bit-reads by bit binding-time

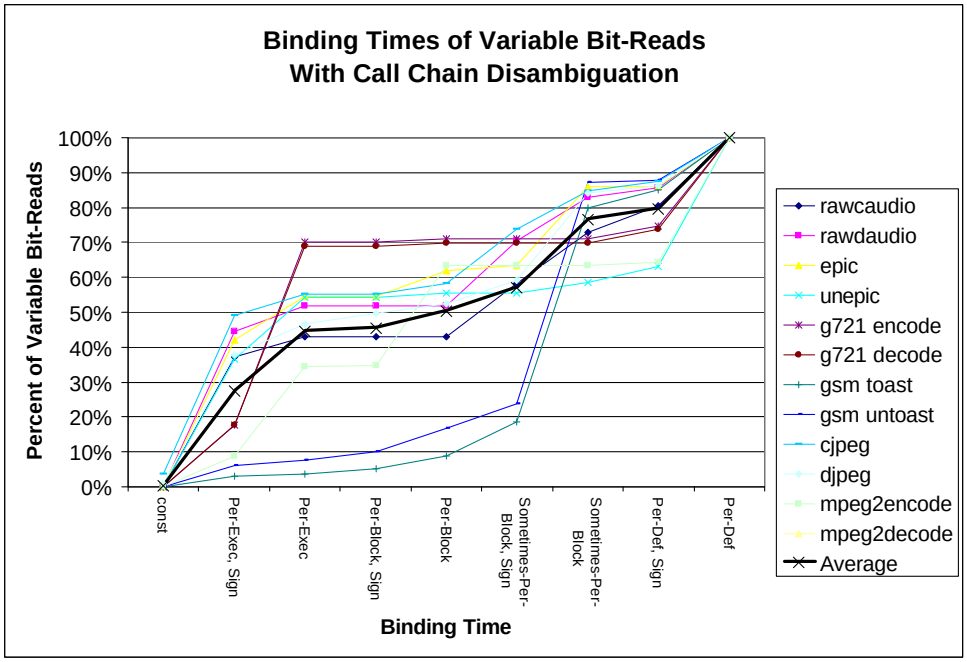


Figure 3.7: Call chain disambiguation: Cumulative distribution of variable bit-reads by bit binding-time

of variable storage, nor on the breakdown of bits by binding-time. Table 3.7, on the other hand, shows that call chain disambiguation can have a significant effect on the access pattern to variables. A few applications experience negligible change, but in the applications that do change, the total number of bit-reads to variables shrinks by about tenfold. This is presumably because the program transformation subsequently enables more effective compiler optimization. The massive reduction in bit-reads is actually the most beneficial outcome of call chain disambiguation. The resulting optimized program, while faster, is actually (relatively) more dynamic and harder to specialize for reasons described below.

Call chain disambiguation and subsequent compiler optimization tends to create more dynamic bit binding-times. The fraction of bit-reads to bits defined per execution (Per-Exec and Per-Exec, Sign) decreases by an average 6.4% (from 50.6% to 44.2%), while the fraction of bit-reads to dynamic bits (Per-Def and Per-Def, Sign) increases by an average 4.9% (from 18.8% to 23.7%). Also, the fraction of bit-reads to sign bits drops across all binding-times, going in total from 56.1% to 37.6%. Nevertheless, it is incorrect to say that call chain disambiguation makes a program more dynamic, since it actually reduces the total bit-read count by up to tenfold. It makes a program *relatively* more dynamic with much reduced activity.

Table 3.8 shows the effect of call chain disambiguation and subsequent compiler optimization on several measures of program size and performance. Execution time is seen to decrease on average to 67.4%, while executable size increases by no more than 50% (except in the case of `mpeg2encode` which expands tenfold; similarly, `mpeg2decode` expanded so much that call chain disambiguation could not be completed). Interestingly, call chain disambiguation tends to increase program size without increasing data size (*i.e.* without adding many variable and heap bits). In some cases, data size actually decreases. This is again testament that the program transformation enables more effective compiler optimizations.

	Relative Size			Relative Performance		
	Executable	Variable Bits	Heap Bits	Execution Time	Variable Bit-Reads	Heap Bit-Reads
rawcaudio	102.5%	100.0%	–	75.7%	100.0%	–
rawdaudio	103.0%	100.0%	–	71.1%	100.0%	–
epic	113.3%	100.0%	100.0%	35.4%	24.7%	100.0%
unepic	88.8%	101.2%	100.0%	100.6%	100.0%	100.0%
g721 encode	134.2%	87.9%	–	82.7%	10.2%	–
g721 decode	153.8%	89.3%	–	82.6%	10.3%	–
gsm toast	152.8%	97.5%	99.8%	40.3%	7.1%	100.0%
gsm untoast	152.8%	99.3%	99.7%	35.2%	3.5%	100.0%
cjpeg	123.7%	96.4%	100.0%	88.3%	91.8%	100.0%
djpeg	113.8%	101.8%	100.0%	82.6%	95.7%	100.0%
mpeg2encode	998.6%	92.7%	100.0%	47.4%	4.1%	100.0%
mpeg2decode	–	–	–	–	–	–
Average	203.4%	96.9%	99.9%	67.4%	49.8%	100.0%

Table 3.8: Call chain disambiguation: Change in size and performance of MediaBench applications. All quantities are relative to without call chain disambiguation.

Although the transformations reduce bit-reads to variables, they leave bit-reads to the heap largely unaffected. This greatly emphasizes heap access while deemphasizing variable access. Thus, the potential benefit of specializing around constant variables is reduced.

3.7 Constancy In Some Lifetimes

Table 3.4 shows that that 15.0% of variable bit-reads in MediaBench are to non-sign Sometimes-Per-Block bits. These are bits that take on a single value in some of their scope instantiations (lifetimes) but more than one value in others.

The question we pose is how often (dynamically) is a bit of this class actually bound once in its scope? If the answer is most of the time, then the bit could be specialized for the common case using the same methods as Per-Block bits. The uncommon case of multiple definitions per block would have to be detected each time through the block and handled accordingly. Such a probabilistic specialization scheme could be profitable if the total cost of handling the uncommon case were smaller than the savings had by specializing the common case. We define

a local variable's *lifetime* as a single pass of control flow through the variable's scope of definition (resulting in creation, use, and destruction of an instance of the variable). The metric we examine is the fraction of a variable bit's lifetimes in which the bit is bound once, *i.e.* is constant through the lifetime.

Table 3.9 and Figure 3.8 show the number of bit-reads to Sometimes-Per-Block bits that are constant in at least 90% and 50% of their dynamic lifetimes. Note, the table does not tally Per-Block bits, which are effectively constant in 100% of their lifetimes. Also, the table only tallies non-sign bits (sign bits are less interesting here since they can be specialized by a one-time narrowing of the data-path). The results shown are for unoptimized versions of the MediaBench applications. Nevertheless, the break-down of bit-reads by binding-times in these results was remarkably similar to the break-down of optimized compilations. The total contribution of bit-reads tabulated here is 14.0%, whereas the total contribution of Sometimes-Per-Block bit-reads in optimized MediaBench applications (Table 3.4) is 15.0%.

The results are not promising for specialization. Among the 14.0% of bit-reads to Sometimes-Per-Block bits, on average only 0.65% are to bits constant in over 90% of their lifetimes. Only 1.7% of bit-reads are to bits constant in over 50% of their lifetimes. This indicates that most Sometimes-Per-Block bits are only rarely bound once per lifetime. Those bits which are frequently bound once per lifetime may be amenable to probabilistic specialization, but their contribution to the total bit-read activity of a program is so small that their specialization would have negligible effect on total run-time.

	Bit-Reads to bits const in >90% Lifetimes	Bit-Reads to bits const in >50% Lifetimes	Total Bit-Reads
rawcaudio	0.61%	1.38%	13.89%
rawdaudio	0.23%	0.69%	11.70%
epic	1.29%	3.85%	9.06%
unepic	–	–	–
g721 encode	0.85%	1.32%	5.36%
g721 decode	0.14%	0.57%	5.26%
gsm toast	0.07%	0.73%	23.78%
gsm untoast	0.01%	0.30%	13.77%
cjpeg	0.12%	0.23%	9.14%
djpeg	0.52%	1.06%	18.49%
mpeg2encode	4.14%	8.28%	28.10%
mpeg2decode	0.47%	2.76%	16.54%
pegwitenc	0.01%	0.34%	14.65%
pegwitdec	0.00%	0.36%	11.89%
Average	0.65%	1.68%	13.97%

Table 3.9: Variable bit-reads to bits constant in some lifetimes (non-sign Sometimes-Per-Block), categorized by fraction of lifetimes in which bit is constant

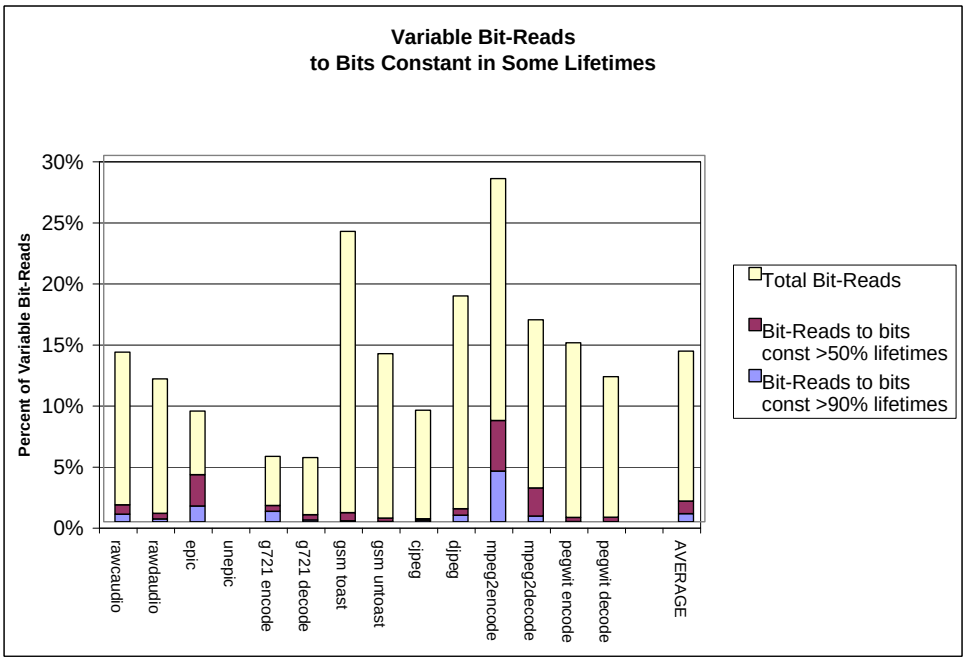


Figure 3.8: Variable bit-reads to bits constant in some lifetimes (non-sign Sometimes-Per-Block), categorized by fraction of lifetimes in which bit is constant

3.8 Sensitivity to Inputs

A natural question for profile-based systems is whether the results of profiling with one input set are indicative of an application's behavior with other inputs. Relative insensitivity to inputs is a desired property for profile-driven specialization. It makes specialization easier and should yield better run-time efficiency. Also, insensitivity to inputs is important for validating the results of this project, since the results reported are typically based on a single data set per application. This section presents preliminary evidence that bit binding-time profiles are largely insensitive to variation in input data sets.

Media processing applications typically run large streams of data through well-defined computational pipelines. It is conceivable that execution profiles for a given application would vary little across different input data files, because the processing pipeline is fairly rigid. Nevertheless, we might find significantly different profiles for different operating modes of an application, because different modes do affect the processing pipeline. For instance, choosing between a compression and decompression mode may select an entirely different pipeline.

With this in mind, we conducted a preliminary study of input sensitivity using the `gzip` 1.2.4 compression program. `gzip` is a freely-available, lossless, LZW-based compression suite. It is a convenient representative application for this experiment for two reasons. First, it implements both a compression and a decompression mode in the same executable, selected by a command-line option. Second, it can operate on any file, so input data sets are plentiful.

Four input files were chosen to represent four different data domains for `gzip`: executable machine code, C source code, English text, and a graphic image. Table 3.10 lists details of those input files. To profile the compression mode, we instrument and run `gzip` on each file. To profile the decompression mode, we run `gunzip` (effectively `gzip -d`) on a compressed version of each file.

File	Content Description	Uncompressed Size	Compressed Size
`vmlinux`[a]	PowerPC machine code (Linux kernel)	1.8MB	684KB
`gzip-1.2.4.tar`[b]	C source-code, tarred	780KB	216KB
`Shakespeare.tragedies`[c]	English text	1.4MB	541KB
`lena.raw`[d]	512x512 8-bit image (Lena's face)	256KB	206KB

[a]From: `ftp://ftp.dodds.net/pub/linux/pmac/vmlinux`
[b]From: `ftp://ftp.cdrom.com/pub/gnu/gzip-1.2.4.tar.gz`
[c]From: `http://ftp.std.com/obi/Shakespeare/Shakespeare.tragedies`
[d]From: `ftp://ftp.cdrom.com/pub/X11/R5contrib/wavelet_pic/lena.raw`

Table 3.10: Data files for gzip input sensitivity experiment

The experiment was conducted early in the project development and thus suffers from two technical flaws. First, it was performed using an early version of the bit binding-time lattice, hence the results presented do not distinguish between sign and non-sign bits. Second, compiler optimizations were not performed prior to instrumenting the application, hence the profile results reflect the unoptimized programmer's model of the computation (not what would be executed on a processor).

Table 3.11 presents the bit binding-times for variables using the four input files described above. Figure 3.9 shows the same results graphically. We find some sensitivity to inputs among the more dynamic bit binding-times but little sensitivity everywhere else. In particular, the largest binding-time class, also the one most promising for specialization— bits bound once per execution— shows little variation. It contributes 50–56% of variable bit-reads across all data sets and operating modes.

There is some variation with operating mode (compression or decompression) in the profile of the most dynamic binding-times. Compared to compression, decompression exhibits on average 10.3% more bit-reads to Per-Def bits and 7.5% fewer bit-reads to Sometimes-Per-Block bits. Thus, decompression seems to have marginally more dynamic behavior.

Within each operating mode, profiles are consistent across all input data files

		Variable Bit-Reads (Millions)	const	Per Exec	Per Block	Sometimes Per Block	Per Def
gzip	`vmlinux`	8895	0.0%	52.3%	4.8%	29.2%	13.7%
gzip	`gzip-1.2.4.tar`	2812	0.0%	52.0%	3.7%	30.0%	14.2%
gzip	`Shakespeare.tragedies`	7699	0.0%	53.5%	4.5%	29.4%	12.6%
gzip	`lena.raw`	898	0.0%	55.1%	3.7%	15.7%	25.5%
gzip	Average	5077	0.0%	53.2%	4.2%	26.1%	16.5%
gunzip	`vmlinux`	1163	0.0%	51.1%	4.0%	20.2%	24.7%
gunzip	`gzip-1.2.4.tar`	411	0.0%	50.1%	2.9%	23.3%	23.8%
gunzip	`Shakespeare.tragedies`	915	0.0%	51.4%	3.0%	18.3%	27.3%
gunzip	`lena.raw`	248	0.0%	56.0%	2.0%	12.8%	29.2%
gunzip	Average	685	0.0%	52.2%	3.0%	18.6%	26.2%
Average		2881	0.0%	52.7%	3.6%	22.4%	21.4%

Table 3.11: Bit binding-times for `gzip` bit-reads using different inputs

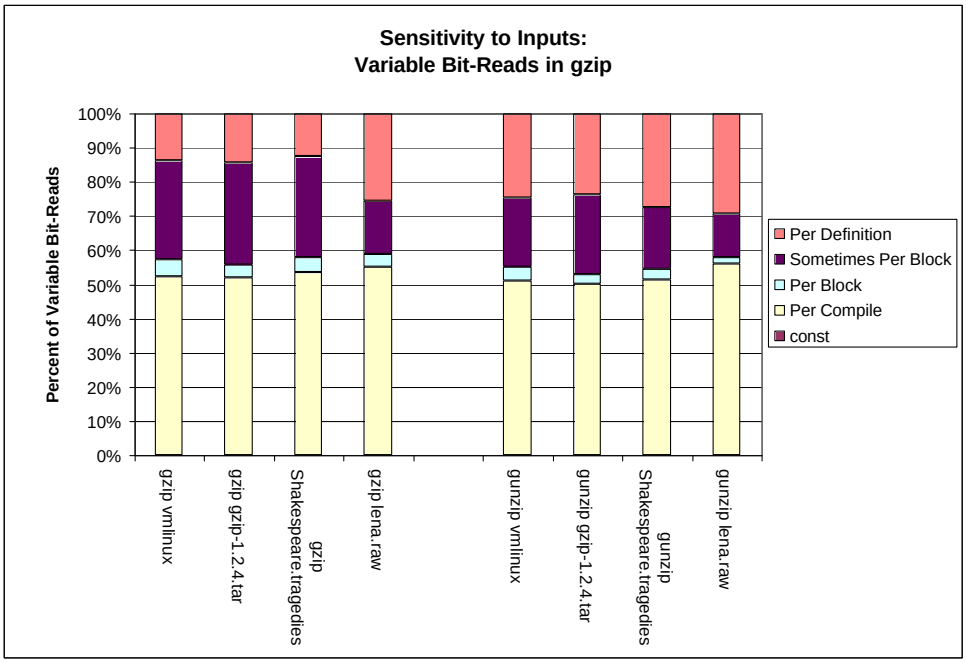


Figure 3.9: Bit binding-times for `gzip` bit-reads using different inputs

except with the graphic image, `lena.raw`. The image file tends to make `gzip` behave more dynamically, pushing some 14% of bit-reads from Sometimes-Per-Block to Per-Def in compression, and pushing somewhat less in decompression. The image file differs from the other inputs in that it is the only format known to be generally incompressible by LZW. In that sense, the image file is a known "bad" input for which high performance is perhaps unimportant. Thus, we may draw a weaker conclusion that the bit binding-time profile for `gzip` is insensitive to data inputs across a large variety of compressible input formats.

Similar findings are reported in the literature that data constancy is largely insensitive to any particular data set. Calder *et al.* [6] report on a variety of data invariance metrics for processor instructions, citing little variation across data sets for Spec95 programs. Fisher and Freudenberger [11] report similar invariance for profile-based branch prediction of Spec programs but concede that every program has "bad" input sets that exercise unusual parts of the code. These findings and our own suggest that a program's constancy behavior is more dependent on algorithmic structure than on particular data values.

3.9 Summary and Conclusions

We have presented a methodology that tallies "bit-read" operations. We consider a read to be any reference in the source code to a named variable or a memory location. In C, such reads are always done on words, *e.g.* on 8-bit "char" types or 32-bit "int" types. Our methodology tallies each such read as a collection of parallel bit operations, *i.e.* bit-reads. The methodology then reports which bit-reads accessed constant bits and which accessed dynamic bits. The categorization of constant vs. dynamic bits is actually on a lattice of binding times representing: (1) the frequency of a bit's change with respect to the scope of its variable, and

(2) whether or not the bit is a sign extension bit.

We find that the constancy of bits on the heap is highly application-dependent. In the applications examined, anywhere from 11.5% to 99.95% of bit-read operations on the heap are to unchanging bits. Although this is not wholly promising, we further find that accesses to the heap are dwarfed by accesses to variables. Hence we subsequently concentrate only on variables.

We find that 70% of all bit-reads in variables are to easily-identified constant bits. This represents a significant raw opportunity for specialization. It means that 70% of all bit operations are doing repeated, wasteful work on constant or previously-seen inputs. Aggressive specialization should be able to drastically shrink the number of required, useful bit operations. Several applicable approaches to specialization are discussed in the next chapter, in Sections 4.4 and 4.5. The breakdown of this 70% follows.

We find that, on average, 56.1% of bit-reads in variables are to sign-extension bits. Sign extension bits are particularly easy to to exploit, simply by using narrower data-paths. An ALU operation (add, subtract) typically needs to use only one sign bit. Hence, it may be possible to remove more than half of a program's bit operations simply by specializing to use narrower data-paths. This is, arguably, the most promising and useful finding of the experiments. The remaining results contribute relatively little opportunity for specialization. They are included for completeness.

We find that, on average, 10.1% of bit-reads in variables are to Per-Exec bits, *i.e.* non-sign bits that remain constant during program execution. It may be possible to specialize the data-paths that process these bits, using bit-level partial evaluation, once during execution, after the bit values are known. However, these bit-reads touch, on average, 38.3% of all bits of a program's variables. Hence they represent a very large collection of constant values, perhaps too large to be

usefully specialized.

We find that, on average, 3.8% of bit-reads in variables are to Per-Block bits, *i.e.* non-sign bits that remain constant during each scope invocation of their variable. These variables represent local variables that are instantiated anew whenever execution enters their block scope (In C, such a block is a curly-braced statement block). It may be possible to specialize the data-paths that process these bits, using bit-level partial evaluation, once on each entry into the block scope. A net performance benefit may be had if the total cost of dynamic respecialization were less than the total savings had by specialized execution. Nevertheless, the contribution of such bits to the total bit-read activity of a program is so small that such specialization is probably not worth the effort.

Beyond the easy 70% of bit-reads, we find that, on average, 15.0% of bit-reads in variables are to Sometimes-Per-Block bits, *i.e.* non-sign bits that remain constant in only some of the scope invocations of their variable. If a given bit were constant in most of the scope invocations of its variable, then it could be specialized as if it were constant in all of the invocations, provided that additional hardware guards were used to detect and special-case those instances when the bit is not constant. Such probabilistic specialization could have an overall net benefit if a bit were constant often enough (in particular, the cost of dynamic respecializations and special-casing must be less than the savings had by specialized execution when the bit is constant). In practice, however, we find that only about 1/20 of the bit-reads in this category (representing 0.65% of bit-reads in variables) are to bits which are constant in more than 90% of the scope invocations of their variable. The contribution of such bits to the total bit-read activity of a program is so small that such specialization is probably not worth the effort.

We find that the final 14.6% of bit-reads in variables are to Per-Def bits, *i.e.* non-sign bits that change dynamically in their variable's scope. These bits,

as well as most Sometimes-Per-Block bits, have patterns of change that do not correspond well to lexical blocks and hence cannot be identified or specialized using the methodology of this study. It possible that some of these bits retain a constant value for long epochs, so that they could in principle benefit from dynamic specialization. A further analysis to discover such cases is discussed in Section 4.3.3.

In addition, we explored a program transformation (call chain disambiguation) intended to discover if a code block behaves differently when called from different call sites, so as to allow separate optimizations/specializations for different call sites. The transformation was found to enable substantial optimizations during standard C compilation, leading to program executions with, on average, 10 times fewer bit-reads, without a substantial increase in code size or data size (note, however, that some programs are not affected at all by this transformation). The remaining code exhibits somewhat less opportunity for specialization. In particular, only 37.6% of all bit-reads in variables are to sign bits (as compared with 56.1% in code without call chain disambiguation). Regardless of its value to bit-level specialization, we find that this program transformation may be of great value in traditional C compilation for microprocessor architectures.

The following chapter (Discussion) proposes further analyses to uncover yet more specializable bit constancy, along with a number of techniques for exploiting that constancy in hardware implementations of C programs.

Chapter 4

Discussion

The experimental results of Chapter 3 (Results) indicate a substantial opportunity for bit-level specialization of C programs. This chapter discusses possible applications of those findings in specializing hardware and software, as well as recounting problems, limitations, and possible extensions to the experimental methodology. We begin in Section 4.1 with a discussion of problems and limitations in the profiling implementation of this study. Section 4.2 in particular discusses the implementation's excessive memory usage and how to control it. Section 4.3 proposes several further analyses to uncover more bit-level constancy. Section 4.4 discusses mechanisms for exploiting the bit-level constancy found by profiling, including possible language extensions and run-time support for dynamic specialization. Section 4.5 explores the specialization of multipliers, proposing a multiply-centric profiling methodology, a cost model, and implementation issues.

4.1 Problems and Limitations

This section discusses problems and shortcomings of the profiling methodology used in this project. We discuss some conceptual limitations of the approach, as well as implementation issues that weakened results or prevented particular appli-

cations from being profiled. A discussion of the implementation's large memory requirements and how to reduce them follows in Section 4.2.

Relevance to Hardware

Perhaps the most serious shortcoming of the profiling approach in this project is that it cannot directly quantify the available benefit of specializing a hardware implementation with metrics such as time, area, or power. In profiling at the bit level, our analysis identifies computations that may benefit from logic-level specialization of hardware data-paths. However, in profiling storage, our analysis is actually abstracted from the specifics of any logic-level, hardware implementation. Finding a constant bit in a storage location does not, in and of itself, indicate the performance benefit that the associated data-paths would get from specializing around that bit's value.

Quantifying the hardware benefit of specializing a program with constant bits requires a model for the hardware cost and specialization techniques of each operation on those bits. Different operations are amenable to different forms of specialization. Section 4.5 develops such models for the multiply operation in particular.

The closest description that one might extract from a C program of the actual operations to be done in hardware is a machine-specific instruction sequence. Alternatively, one could use a machine-independent sequence such as SUIF's internal program representation. Bit constancy would then be profiled in instruction operands rather than storage locations. Analyzing an instruction stream may be appropriate for evaluating alternative microprocessor hardware, *e.g.* moving to a SIMD multimedia ISA with a vector of narrow ALUs. However, an instruction stream introduces ISA-specific grouping and sequentialization of primitive operations. So it is still not ideal for evaluating savings from custom hardware, where

primitive operations may be grouped and parallelized more flexibly.

Aggregate Data Structures

The profiling methodology of this project is geared to integer data types. The use of sign bits in the bit binding-time domain assumes that every word is in one's- or two's-complement representation. In the present implementation, aggregates such as C `structs` and arrays are analyzed as very wide integer objects. This completely ignores the structure of individual words in the aggregate and produces faulty information on sign bits. A type-aware analysis is needed to decompose aggregates into constituent integer elements and to capture their respective bit binding-times.

Recognizing which component of an aggregate is being accessed is simple for direct variable access but difficult through a pointer. In direct variable access, the component is identified either by its explicit name (*e.g.* `x.y`) or its offset (*e.g.* `x[y]`). If the component is accessed through a pointer, then the profiling library knows only which run-time address is being accessed. The *core* can identify which object contains that address (see Section 2.3.1). Identifying which component of an object is being accessed further requires knowing the type of the object. This is easy enough if the object turns out to be a variable, since its type would be known statically from the C source code.[1] However, it is in general impossible to know the type of an object on the heap. The heap allocation routine used in C, `malloc`, is not strongly typed and knows nothing about the allocated object except for its size. The only clue as to the type of an object on the heap comes from the type of the pointer used to access it. A complication arises if an aggregate is accessed by several pointers of *different* types (*e.g.* a region of memory being block-copied

[1]Identifying which component of an aggregate is being accessed through a dereferenced pointer is not possible for a `union` variable, since multiple types exist for certain offsets in the aggregate

as an array of chars by the construct: `while (a<e) *b++=*a++;`). This multiple typing problem exists with variables as well as heap objects. It is similar in nature to the multiple typing evident in `union` aggregates but exists specifically because of pointer aliasing.

One solution for determining the type of an integer component accessed through a pointer is to always believe the type of the pointer. A `char` pointer must be accessing a `char`, an `int` pointer must be accessing an `int`, and so on. This solution may misinterpret the location of sign extension bits and bit regions within the accessed object, since it is possible to access a given byte of storage in different alignments within larger integer accesses (*e.g.* a byte at address A+`0x3` may be part of a 16-bit `short` at address A+`0x2` or a 32-bit `int` at address A). This complication is perhaps not an issue for C programs with highly structured, strongly typed conventions, where most bytes of storage are only ever accessed as one type.

4.1.1 Problems with SUIF

The SUIF compiler proved to be a stumbling block for profiling some applications. In some cases, the SUIF front end could not be made to accept a program. In some cases, the SUIF optimization passes crashed. In some cases, the SUIF back-end emitted a faulty program which would subsequently crash. These problems prevented a number of programs from being analyzed. Here we list those programs along with the specific problem encountered:

- Programs from UCLA MediaBench suite: [29]

Ghostscript	SUIF linker fails
Pegwit, PGP, RASTA	SUIF optimizations (`porky`) run out of memory

- Programs from SPECint95: [40]

m88ksim	SUIF C back-end crashes due to variable-length argument lists (`va_list`)
compress, li	Emitted program crashes
ijpeg	SUIF C back-end crashes
perl	Procedure duplication for call chain disambiguation fails (too many copies of `safemalloc()`)

Certain SUIF optimizations proved to be frequently buggy and were not used on any profiled applications. These include:

- `-fold` — constant folding
- `-reduction` — reduction to move summation out of a loop
- `-ivar` — induction variable detection and reduction

The task of running an application through the SUIF-based transformation and compilation pipeline is a difficult one. It is not possible to simply substitute the SUIF compiler in place of a traditional C compiler. Each SUIF pass (front end, linking, instrumentation, back-end, *etc.*) requires a separate program execution. Some passes operate on individual source files, like traditional incremental C compilation, while some passes process the entire file set at once. Hence it is not sufficient to use traditional Makefile rules that simply map one file extension into another. Instead, each application requires a new, custom Makefile with an explicit sequence of SUIF passes (Appendix B explains this compilation sequence). For some applications with sophisticated Makefiles and build scripts (notably `gcc` from SPECint95), identifying and/or creating a working set of source files and creating a new Makefile proved to be so difficult that the applications were forsaken

altogether.[2]

4.2 Reducing Memory Usage

One major shortcoming of the profiling methodology of this project is its large memory requirement. As seen in Section 2.3.3, the analysis expands both code and data sizes, leading to a memory requirement as high as 100-300 times that of the original program. The expansion of data memory is perhaps a factor of 2-3 times larger than necessary due to a poor implementation of memory and variable images. However, the only promising way to significantly reduce the total memory requirement is to selectively profile only a small fraction of the program's storage. As the 90-10 rule suggests, one should be able to capture most of the interesting activity in a program with only a small fraction of the storage. Here we discuss several ways of choosing which storage to profile and which to ignore.

Profile kernels only This two-pass strategy seeks to profile only those lexical blocks that comprise computational kernels, *i.e.* that contribute a large fraction of the program's execution time. A first execution is needed to identify kernels, using a profiling tool such as `gprof` [18]. A second execution is instrumented to profiles the kernels.

Profile "best" variables only This two-pass strategy seeks to profile only those variables that excel at some particular metric, *e.g.* highest read count. A first execution is needed to compute the desired metric for all variables (it is necessary that computing the desired metric require less memory than the conventional bit binding-time analysis, or else no savings are realized). A

[2]The SUIF community is quick to point out that SUIF1 *cannot* compile the SPEC95 suite. Harvard's *MachSUIF* extensions introduce bug fixes to enable compilation of SPEC95. We use SUIF1 with MachSUIF but were still unable to successfully instrument and compile SPEC95. SUIF2 (presently in beta) also promises to compile SPEC95.

second pass is instrumented to profile only the variables with best metric. Note that the metric does not have to be a variable's read count but should rank the variable's importance or cost in the implementation.

Cache "best" variables Rather than preselect which variables to profile, a cache model could be used to select variables on the fly. In this one-pass approach, the profiling system reserves a fixed amount of memory to fully profile a small number of variables. Variables enter and leave this cache according to a metric-based eviction policy. The most appropriate metric is probably a variable's read count. Additional memory may be needed to compute this metric for all variables. The cache model assumes that a particular working set of most frequently read variables will emerge during execution. As with any cache, thrashing is possible. One disadvantage of this scheme is that a profiled variable may not spend the duration of the program in the cache, hence its binding-time information will be incomplete. Cache contents could be periodically analyzed and written to file in order to catch an evolving working set of variables.

Additional savings in memory may be had by collapsing array variables, *i.e.* by keeping incremental binding-time information for only a single, representative element. This is discussed in Section 4.3.1.

4.3 Further Analyses

Several transformations and analyses were considered in detail but never implemented due to lack of time. Their purpose is collectively to uncover more forms of bit constancy. These approaches are discussed here.

4.3.1 Collapsing Array Information

At present, the profiling system analyzes each array as a single, very wide integer. As discussed in Section 4.1, this makes the results for arrays less useful and possibly erroneous, since sign bits are not properly identified. A useful way to analyze an array in this profiling framework is to combine the bit binding-times from each array element by least-upper-bound (LUB) into a single representative element. We refer to this as *collapsing* the array.

Collapsing an array into a single element is not likely to lose any significant element-wise information. This is because array elements are seldom accessed individually. Media processing programs in particular tend to walk sequentially over ranges of array elements, often over an entire array. Thus in a first order analysis, we expect that all elements of an array have similar constancy. Even when this is not true, a representative element formed by a LUB is conservative, and any specialization based on it will be computationally correct.

The actual collapsing of elements can be done at one of several points in the profiling framework. One approach is to collapse at program exit, while computing the result report. This requires incrementally computing and storing binding-times for each array element during execution. An alternative approach is to incrementally compute binding-times only for the representative element. This approach can save significant memory, because the array's memory image can be simplified to contain only the most recent value (it is possible to avoid even this cost[3]). A full memory or variable image containing counters and bit masks would be needed only for the representative element.

[3]It is possible to avoid keeping a most recent value for each storage location. This requires each write operation to be profiled immediately prior to the write, when the previous value is still in place. However, this approach will not recognize value changes made by unprofiled code, *e.g.* by a library routine.

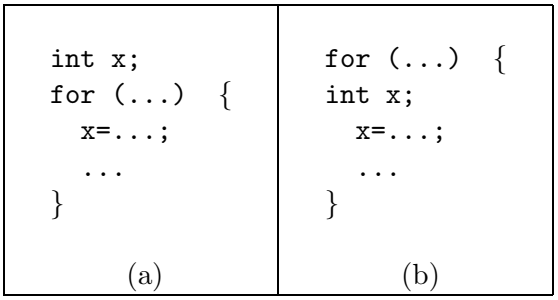


Figure 4.1: Pushing a variable declaration to the point of use

4.3.2 Pushing Variable Declarations to Point of Use

Declaring a variable well before its use may make that variable's lexical scope appear needlessly large and its binding-time needlessly weak. The bit binding-time of a variable bit measures its rate of change with respect to the variable's scope of definition. That binding-time can be thought of as covariant with constancy but contravariant with the size of the block. For a tighter analysis, it is beneficial to identify for each variable the smallest block in which that variable could possibly be declared.

Consider, for instance, a variable `x` that is updated in each iteration of a loop and does not embody a loop-carried dependency (*i.e.* its value from one iteration is not used in subsequent iterations). Its logical scope is the body of the loop. In Figure 4.1(a), the variable is declared outside the loop body. Because it changes value during its lifetime, its bits are classified as Per-Def. In Figure 4.1(b), the variable is declared inside the loop body. Now it is constant in each lifetime, so its bits are classified as Per-Block. Moving the variable declaration into the smallest possible scope yields a stronger binding-time, which is more useful for a specialization system.

It is easy to formalize and automate the process of identifying the smallest scope in which a variable could be declared. One could write a SUIF compiler pass to identify that smallest scope and to transform the program so as to push

the variable's declaration to that point.[4] At best, such a pass will promote the classification of certain bits from Per-Def and Sometimes-Per-Block to Per-Block.

4.3.3 Bitwise Reads-Per-Write, Reads-Per-Change

The bit binding-time domain does not detect slowly-varying behavior that is not clearly related to a lexical block. For instance, it cannot detect when a variable is constant in one loop and dynamic in another loop. If the first loop runs for very many iterations, it may be profitable to specialize it around the variable's constant value. A variety of source-level transformations can go some way to disambiguate a value's scope of constancy to the profiling methodology. Pushing variables to the point of use (Section 4.3.2) helps by refining a variable's lexical scope. A transformation to static single assignment (SSA) form would add flow sensitivity, *i.e.* disambiguate the use of a variable in different places, by splitting it into separate variables. Similarly, Call chain disambiguation (Section 2.5) adds context sensitivity, *i.e.* disambiguates the use of a variable from different call sites, by duplicating the variable and its surrounding code.

One shortcoming of the binding-time model that cannot be ameliorated by any of the above transformations is that it cannot quantify numerically how often a value changes. The transformations help refine the lexical scope of a value, but that scope says nothing about how many times the value is actually used or changed. Specializing around a constant value is only profitable if that constant value is reused many times before changing. Hence we propose two metrics that are orthogonal to lexical binding-time: *reads between writes* (RBW) and *reads between value changes* (RBVC). RBW counts the number of uses a variable gets between write operations. RBVC counts the number of uses a variable gets between writes that actually change the variable's values. These metrics help

[4] A SUIF pass to push variable declarations into the scope of use was actually written but not fully debugged. Its results are not discussed in this report.

identify, for instance, which Per-Block bits are actually worth specializing once per block, and which Per-Def bits are worth specializing ever.

Measuring average reads between writes and average reads between value changes is actually trivial. It is equivalent to computing the ratio of total reads to total writes and total reads to total value changes, respectively.[5] The former requires read and write counters for each variable, which are already implemented. The latter requires a change counter for each bit, which is a rather large overhead (a 64-bit change counter would make for 64 bytes of overhead per byte of original program storage). Measuring the standard deviation of these metrics is not much more complicated, since it can be computed incrementally using several additional counters. The value of these metrics may in fact vary at different times during program execution. Thus, one might consider reporting the averages multiple times during execution.

4.4 Exploiting Bit Constancy

The goal of the analyses in this paper has been to quantify computational waste in the form of constant and slowly-varying bits. Ultimately, the goal is to find ways of avoiding or exploiting that waste. There are at least two distinct approaches to that end. One approach is to modify the language and program representation to explicitly reduce computational requirements. Another approach is to transparently modify the implementation, specializing either the hardware or the software around known bit behavior. In this section we pose some thoughts about both approaches.

[5] To compute the mean reads-between-writes (RBW), suppose that for each write w_i ($i \in \{1...N\}$) we record the number of reads r_i which occured between writes w_{i-1} and w_i. Then $RBW = \text{mean}(r_i) = \frac{1}{N}\sum_i r_i = \frac{\text{total reads}}{\text{total writes}}$. Similarly, the mean reads-between-value-changes $RBVC = \frac{\text{total reads}}{\text{total value changes}}$.

4.4.1 Language Features

The results of this study suggest that there are two language features which may help exploit constant bits: run-time constants and explicit bit widths.

A dynamically valued constant (DVC) is variable whose value is constant but not actually known until run-time. Declaring a DVC in effect allows the programmer to explicitly declare the binding-time of a variable and the lifetime of its value. C++ can declare DVCs using the `const` keyword, which declares that a local variable's value will be constant in its scope of definition but may be different the next time the scope is entered. With respect to our bit binding-times, a DVC is a variable all of whose bits are bound Per-Exec or Per-Block (including sign variants). We have already seen in Section 3.5 that, in MediaBench, entirely constant (Per-Exec) words (a subset of DVCs) account for 7.1% of all bit-reads in variables. More general DVCs account for still more bit-reads. One may be tempted to add a language feature to define individual bits of variables as DVCs. However, this level of expressivity is cumbersome and error-prone.

The second language feature, explicit bit widths, would allow a programmer to explicitly request narrow width computation. In C, as in most high level languages, data widths are quantized at certain powers of 2 (typically 8, 16, 32, and 64 bits.) If a variable needs just one extra bit (*e.g.* to detect overflow), its width must be doubled to the next available size. Sections 3.4 and 3.5 support this notion in the finding that nearly half of all bit-reads in variables are to high-order regions of constant (Per-Exec and Per-Exec, Sign) bits. If word data types contained explicit bit widths, a compiler could infer the minimum allowable precision (bit width) of every computation and result (*e.g.* 12-bit + 3-bit = 13-bit, using a 12-bit adder). Alternatively, the programmer could explicitly state the bit width of each primitive operation (*e.g.* 12-bit + 3-bit should be done using a 3-bit adder, and the programmer guarantees that values will not overflow). However, explicit

operation widths are cumbersome to the programmer and highly error-prone.

4.4.2 Exploitation in Hardware

Given the position of constant bits in a variable, there are three possible cases for specialization: (1) entirely constant words, (2) constant high-order or low-order bits, and (3) constant bits in random positions. Specialization may be considered in each of these cases regardless of the actual binding-time of bits. The bit binding-times only determine *when* specialization should take place.

A computation with constant word operands (all Per-Exec and/or Per-Block bits) might be folded in its entirety. This is possible in any kind of hardware implementation, microprocessor or custom logic. If only some of the word operands are constant, then a custom hardware implementation might still be specialized by partial evaluation, *e.g.* a constant coefficient multiplier [33] [9] (a constant times a dynamic).

A computation whose operands have constant high-order or low-order bits may merit a narrow data-path implementation. This is appropriate for a custom hardware implementation where a data-path of arbitrary width may be synthesized. It may also be possible on a microprocessor if a narrower ALU operation is available and has some advantage, *e.g.* lower power. One interesting use of narrow ALU operations is in the SIMD vector units associated with multimedia instruction sets (*e.g.* Vis [28], MMX [36], AltiVec [35]). Brooks and Martonosi [4] describe one approach to automatically pack operations with constant high order bits into the parallel slots of a SIMD vector instruction.

Finally, a computation whose operands have constant bits in random locations could be specialized in custom hardware by logic optimizations, *e.g.* bit-level constant propagation and folding. This will mostly likely result in a minor area savings and may in fact have an adverse effect on the regularity of a data struc-

ture.

4.4.3 Probabilistic Specialization

Bit binding-time information collected by profiling is necessarily data dependent and may not be universally applicable. If the constancy behavior of a program is in fact sensitive to data, then any specialization based on one data set may not be appropriate for another data set. For example, a narrow computation appropriate for one data set may overflow with another data set and affect program correctness. Section 3.8 indicates that bit constancy in the `gzip` compression program is not very sensitive to the input file. In general, however, one must be prepared for sensitivity and ensure program correctness.

Specializing around profiled information must be done with safeguards. We can probabilistically specialize for the common case but must also detect and handle the uncommon case. Detecting the uncommon case amounts to detecting when the assumptions on input bit behavior are violated for a particular specialized data-path. If a specialization assumes that certain input bits are constant, then a comparator must be used on those bits to verify their value. If a narrow data-path specialization assumes that inputs have a certain width, then comparators or overflow detection must be used to verify that. Hardware support for such verification in microprocessors is rare. One possible mechanism to ensure that a storage location stays constant is to write-protect it using virtual memory techniques [12], so that any write is detected and the written value can be verified. This technique may require placing the storage element on its own memory page, leading to a substantial storage overhead.

A variety of actions might be taken to handle the uncommon case when an input assumption is violated. If a microprocessor is available, it may be possible to handle the uncommon case in software. This is true regardless of the common-

case implementation, be it software, reconfigurable logic, or custom ASIC. The appearance of an uncommon case may provide a signal that the common-case implementation is too specific and that a more general implementation should be used for all subsequent computations. In a reconfigurable logic implementation, we might consider reloading a slightly less specialized configuration to handle all subsequent computation. Depending on the relative cost of switching out to the uncommon-case implementation and back, it may or may not be beneficial to return to a specialized implementation after such an interruption.

4.4.4 Online vs. Offline

The complementary processes of profiling and specializing an application may each be done online (in real-time) or offline. The profiling methodology described thus far can be considered offline, since its large memory and run-time requirements make it impossible to incorporate it into a casual execution of a program.

Any online profiling methodology must contribute only negligible overhead to a program's memory consumption and run-time. Modifying our profiling methodology into an online version would involve, first and foremost, implementing the memory savings techniques described in Section 4.2. Additional run-time savings might be realized by profiling during only part of the execution. The `gprof` profiling system [18] does this by profiling only on interrupts. As a coarse solution, we might consider profiling snapshots of storage at interrupts. Alternatively, we could periodically switch a program between self-profiling and non-profiling modes. This would require patching the program to have both profiling and non-profiling versions of each routine, and selecting which version to run using jump tables.

A continuum of specialization approaches exists, including online, offline, and combinations thereof. Purely online specialization is probably too expensive to

consider, since it requires online recompilation or logic synthesis in addition to online profiling. Purely offline specialization, including offline profiling, is perhaps the easiest to implement. One attractive hybrid approach might involve the offline generation of several specialized instances of a kernel routine, then the online selection of a particular instance depending on the constancy behavior detected in that execution by online profiling.

4.5 Application: Specializing Multipliers

The precise benefit of specialization depends not only on the program's bit binding-time profile but also on the particular operations being specialized. Different operations (*e.g.* add, multiply, *etc.*) admit different mechanisms for specialization and have different cost models for the specialized area and performance. This project uses a simplified profiling model that intentionally avoids distinguishing particular operations. This section considers extensions to the profiling methodology for assessing and specializing a particular operation, namely integer multiplication.

Multiplication is a simple but non-trivial operation to specialize. It has a potentially large area benefit from specialization around narrow width operands, since a spatial $n \times m$-bit multiply has area $O(nm)$. In addition, multiplication is amenable to specialization around only a single constant operand (constant coefficient multiplier [33] [9]).

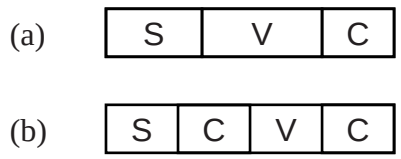


Figure 4.2: Decomposition of a word into bit ranges, (a) SVC model, (b) SCVC model.

4.5.1 A Simpler Constancy Model: Bit Regions

Section 3.5 presented evidence that a large part of the bit-level waste in MediaBench appears in sign bits and in contiguous regions of constant bits in high- or low-order bit positions. This suggests that the binding-time model presented in Section 1.4 could be simplified to consider contiguous bit regions rather than individual bits. Figure 4.2 shows two possible models in which a word is analyzed as a group of bit regions. The region labels shown are:

S — sign extension bits.

These are high-order bits that carry the same value as one another. They can be represented by a single sign bit to create narrower arithmetic data-paths.

V — variable (*i.e.* dynamic) bits.

These are dynamic bits with no a-priori known value.

C — constant (*i.e.* static) bits.

These are unchanging bits whose value is known and can be specialized by partial evaluation of the logic.

The SVC model of Figure 4.2(a) is particularly simple, admitting only sign bits at the high bit positions. The SCVC model of Figure 4.2(b) is a possible extension to allow constant bits in the high bit positions. It is possible for a bit region to have zero width, so SVC may in practice collapse into SV (range-limited around zero), and SCVC may collapse into CV (range-limited around a non-zero value). In the rest of this section, we consider only the SVC model of Figure 4.2(a).

To see how a multiplication may be specialized in the SVC model, consider the following decomposition:[6]

$$\begin{aligned}\mathsf{S_1V_1C_1} \times \mathsf{S_2V_2C_2} = (\mathsf{S_1} \times \mathsf{S_2}) + (\mathsf{S_1} \times \mathsf{V_2} + \mathsf{S_2} \times \mathsf{V_1}) + (\mathsf{S_1} \times \mathsf{C_2} + \mathsf{S_2} \times \mathsf{C_1}) \\ + (\mathsf{V_1} \times \mathsf{V_2}) + (\mathsf{V_1} \times \mathsf{C_2} + \mathsf{V_2} \times \mathsf{C_1}) + (\mathsf{C_1} \times \mathsf{C_2})\end{aligned}$$

The component multiplications may be specialized into the following run-time structures:

$\mathsf{S} \times \mathsf{S}$	—	a single-bit XOR
$\mathsf{S} \times \mathsf{V}$	—	controlled arithmetic negation
$\mathsf{S} \times \mathsf{C}$	—	controlled arithmetic negation of a constant, implemented by word selection (bit-parallel multiplexers or AND/OR plane)
$\mathsf{V} \times \mathsf{V}$	—	a conventional multiplier
$\mathsf{V} \times \mathsf{C}$	—	a constant-coefficient multiplier (ANDs and adders)
$\mathsf{C} \times \mathsf{C}$	—	precomputed to a constant

The only part of the multiply that cannot be specialized is $\mathsf{V} \times \mathsf{V}$. The area and time benefit of specialization depends on the actual widths of the bit regions (minus the cost of producing the specialization).

Note that the SVC model does not incorporate scope-based binding-times. Nominarlly, we consider V to be fully dynamic, *i.e.* bound per-definition, and C to be bound per-compile or per-execution. It is possible to identify V as a slowly-varying value regardless of scope using dynamic metrics such as uses per value change (equivalent to $RBVC$ from Section 4.3.3).

[6] In the notation, adjoining letters represent bit concatenation, not multiplication. Multiplication is represented by $\times$.

4.5.2 Profiling Multiplies

Specializable multiplies may be discovered by profiling the operands of multiply instructions. This is an instance of instruction-based profiling, which is qualitatively different than the storage-based profiling used in this project (the differences are discussed in Section 1.3.3). This approach can still be implemented by instrumentation of SUIF's intermediate form.

The objective of profiling is to discover the SVC profile for both operands of each multiply instruction. This can be done incrementally with minimal storage requirements. For each operand of each multiply, the profiling system must maintain a most-recent value and an incremental SVC profile. The SVC profile can be represented compactly by the bit-positions of the V bit-range, for instance using an LSB/MSB integer pair, or using a bit-mask. At each dynamic execution of a multiply, and for each operand, the profiling system would XOR the operand's present and most-recent values to discover changes, then update the incremental SVC profile by widening V appropriately. An operand's initial profile must be marked as undefined. At the first use of that operand, the profile would be set to have maximal S and C regions, with an implicitely zero-width V.

4.5.3 Specialization models

A practical specialization scheme must decide which multiplies are worth specializing and precisely when to specialize them. In the simplest model, we consider specializing a multiplier only once per execution (or per compile). If the V value is slowly varying, we can consider dynamically respecializing a multiplier around different V values. It is also be possible, in principle, to detect and specialize around slowly-varying precision of operands, *i.e.* for varying V widths. However, this requires significantly more expensive profiling and will not be discussed here.

Specializing Once

In this model, we consider V to be fully dynamic and C to be fully static. If the C values of both operands of a particular multiply are known a-priori (before execution, *i.e.* bound per-compile), then the multiplier could be specialized offline, once and for all. If the C value of either operand will not be known until execution time (*i.e.* bound per-execution), then the multiplier could be specialized online as soon as the values are known. This may be as early as the first dynamic execution of the multiply (possibly earlier with a backwards data-flow analysis) or could be delayed until several multipliers are ready for specialization. A specialized multiplier should include input comparators to ensure that its operands maintain their expected values.

Because the benefit of specialization (especially online specialization) is offset by the cost of producing a specialization, it may not be desirable to specialize all multipliers. The savings of a particular multiplier can be calculated roughly as:

$$\text{savings} = (\text{original cost} - \text{specialized cost}) \times \text{uses} - \text{cost of specialization}$$

where "uses" is the number of dynamic executions of the multiplier.

We must be more precise to obtain comparable units of "cost." The cost of producing a specialization is typically in time (on a microprocessor), but the savings in multiplier cost are more complex. A specialized multiplier may have time advantages such as lower latency and higher throughput, but in a pipelined, feed-forward computation, these time measures are irrelevant (throughput becomes one-per-cycle, and latency is largely ignored). The real benefit comes from reduction in multiplier area, which allows the packing of more parallel hardware for better overall throughput. One heuristic way to express the area savings in time units is to scale a time measure (*e.g.* latency) by the area reduction factor (this is

an idealized savings, pretending that the entire computation consists of identical multipliers). Thus the heuristic time savings of a specialized multiplier are:

$$T_{\text{saved}} = \left(T_{\text{orig}} - T_{\text{spec}} \frac{A_{\text{spec}}}{A_{\text{orig}}} \right) \times \text{uses} - T_{\text{specialization-cost}}$$

where A is area, T can be taken as latency, and "orig" and "spec" refer to the original and specialized multipliers, respectively. A specialization system can then choose the most profitable multiply instructions to specialize by ranking their T_{saved}, post profiling.

Specializing for Slowly Varying Values

If the V (dynamic) part of a multiply operand changes value infrequently, an aggressive system may wish to dynamically respecialize around each new value. Each such specialization treats V as a constant, converting the SVC operand into SC'. It is possible but not necessary for both operands of a multiply to be specialized this way.

The run-time benefit of respecialization for a particular multiply depends on the number of dynamic executions (uses) of each specialized instance. The heuristic savings metric described above is still valid but must now be applied for each specialized instance individually. If the run-time system respecializes at each change of operand value, then the total savings of a specialized multiplier are:

$$T_{\text{saved}} = \sum_{i=1}^{\text{changes}} \left(T_{\text{orig}} - T_{\text{spec}(i)} \frac{A_{\text{spec}(i)}}{A_{\text{orig}}} \right) \times \text{uses}_{(i)} - \text{changes} \times T_{\text{specialization-cost}}$$

Computing this cost precisely is difficult, since it requires remembering the bit pattern and number of uses for every unique V value of every multiplier (the bit pattern is needed to ascertain a precise area-time cost). If we instead con-

sider average costs T_{spec}, A_{spec} over all possible values,[7] then the savings formula becomes:

$$\begin{aligned} T_{\text{saved}} &= \sum_{i=1}^{\text{changes}} \left(T_{\text{orig}} - T_{\text{spec}} \frac{A_{\text{spec}}}{A_{\text{orig}}} \right) \times \text{uses}_{(i)} - \text{changes} \times T_{\text{specialization-cost}} \\ &= \text{changes} \times \left(T_{\text{orig}} - T_{\text{spec}} \frac{A_{\text{spec}}}{A_{\text{orig}}} \right) \times \text{uses} - \text{changes} \times T_{\text{specialization-cost}} \\ &= \text{changes} \times \left[\left(T_{\text{orig}} - T_{\text{spec}} \frac{A_{\text{spec}}}{A_{\text{orig}}} \right) \times \text{uses} - T_{\text{specialization-cost}} \right] \end{aligned}$$

Computing this cost is significantly easier, since it only requires the profiler to collect the total number of changes and uses for each multiplier operand. Again, a specialization system should rank all multiply instructions by savings metric and specialize only the most profitable instructions.

[7] The cost of a constant-coefficient multiplier depends on the coefficient value. Given an SVC profile, an average cost can be computed over all possible values. An average cost over all possible bit patterns can be tabulated for any SVC value. Alternatively, if the profiler collects all values actually used with a multiplier, the average cost can be computed over that set of values.

Bibliography

[1] Tito Autrey and Michael Wolfe. Initial results for glacial variable analysis. In *Proc. Ninth International Workshop on Languages and Compilers for Parallel Computing (LCPC '96)*, pages 120–134, San Jose, California, August 8–10, 1996.

[2] William Blume and Rudolf Eigenmann. Symbolic range propagation. In *Proc. Ninth International Parallel Processing Symposium*, pages 357–363, Santa Barbara, California, April 25–28, 1995.

[3] Kiran Bondalapati and Viktor K. Prasanna. Dynamic precision management for loop computations on reconfigurable architectures. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '99)*, pages 249–258, Napa Valley, California, April 21–23, 1999.

[4] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Fifth International Symposium on High Performance Computer Architecture (HPCA '99)*, Orlando, Florida, January 9–12, 1999. Available as: `http://www.ee.princeton.edu/~mrm/papers/hpca99.pdf`.

[5] Mihai Budiu and Seth Copen Goldstein. Detecting and exploiting narrow bitwidth computations. In *Second Annual CMU Symposium on Computer*

Systems (SOCS-2), Pittsburgh, Pennsylvania, October 2, 1999. Available as: `http://www.cs.cmu.edu/~mihaib/research/socs2.ps.gz`.

[6] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proc. 30th International Symposium on Microarchitecture (MICRO-30)*, pages 259–269, Research Triangle Park, North Carolina, December 1–3, 1997.

[7] Timothy J. Callahan, John Hauser, and John Wawrzynek. The garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, April 2000.

[8] Eylon Caspi. Binding time analysis for bits. Project report for course CS263, *Design and Analysis of Programming Languages*, spring 1998, University of California, Berkeley, May 1998. Available as: `http://www.cs.berkeley.edu/~eylon/cs263/report.ps.gz`.

[9] Kenneth David Chapman. Fast integer multipliers fit in fpgas. *EDN*, 39(10):80, May 12, 1993.

[10] Charles Consel and François Noël. A general approach for run-time specialization and its application to c. In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 145–156, St. Petersburg Beach, Florida, January 21–24, 1996.

[11] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proc. Fifth International Conference on Architectural Support for Programming Language and Operating Systems(ASPLOS-V)*, pages 85–95, Boston, Massachusetts, October 12–15, 1992.

[12] Virtual Memory Primitives for User Programs. A.w. appel and kai li. In *Proc. Fourth International Conference on Architectural Support for Programming*

Language and Operating Systems (ASPLOS-IV), pages 96–107, April 8–11, 1991.

[13] Freddy Gabbay and Avi Mendelson. Can program profiling support value prediction? In *Proc. 30th International Symposium on Microarchitecture (MICRO-30)*, pages 270–280, Research Triangle Park, North Carolina, December 1–3, 1997.

[14] David Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '95)*, pages 136–144, Los Alamitos, California, April 19–21, 1995.

[15] Maya Gokhale and Edson Gomersall. High level compilation for fine grained FPGAs. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pages 165–173, Napa Valley, California, April 16–18, 1997.

[16] Maya B. Gokhale and Janice M. Stone. NAPA C: Compiling for a hybrid RISC/FPGA architecture. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 126–134, Napa Valley, California, April 15–17, 1998.

[17] K.L. Gong and L.A. Rowe. Parallel MPEG-1 video encoding. In *1994 Picture Coding Symposium (PCS '94)*, Sacramento, California, September 1994. See also: `http://bmrc.berkeley.edu/frame/research/mpeg/`.

[18] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *ACM SIGPLAN Notices*, 17(6):120–126, June 1982.

[19] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-directed run-time specialization in C. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pages 163–178, Amsterdam, The Netherlands, June 12–13, 1997.

[20] B. Gunther, G. Milne, and L. Narasimhan. Assessing document relevance with run-time reconfigurable machines. In *Proc. IEEE Workshop on FPGAs for Custom Computing Machines (FCCM '96)*, pages 10–17, Napa Valley, California, April 15–17, 1996.

[21] The gzip home page. `http://www.gzip.org`.

[22] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[23] John R. Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pages 24–33, Napa Valley, California, April 16–18, 1997.

[24] Simon D. Haynes and Peter Y. K. Cheung. A reconfigurable multiplier array for video image processing tasks, suitable for embedding in an fpga structure. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 226–234, Napa Valley, California, April 15–17, 1998.

[25] Luke Hornof and Jacques Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pages 63–73, Amsterdam, The Netherlands, June 12–13, 1997.

[26] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[27] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg Beach, Florida, January 1986.

[28] Leslie Kohn, Guillermo Maturana, Marc Tremblay, and A. Prabhuand G. Zyner. The visual instruction set (VIS) in ultraSPARC. In *Forty-First IEEE Computer Society International Conference (COMPCON '95) Digest of Papers*, pages 462–469, San Francisco, California, March 5–9, 1995.

[29] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. 30th International Symposium on Microarchitecture (MICRO-30)*, pages 330–335, Research Triangle Park, North Carolina, December 1–3, 1997.

[30] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, pages 137–148, Philadelphia, Pennsylvania, May 21–24, 1996.

[31] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proc. 29th International Symposium on Microarchitecture (MICRO-29)*, pages 226–237, Paris, France, December 2–4, 1996.

[32] Mikko H. Lipasti, Christopher B. Wilkerson, and John P. Shen. Value locality and load value prediction. In *Proc. Seventh International Conference on Architectural Support for Programming Language and Operating Systems*

(ASPLOS-VII), pages 138–147, Cambridge, Massachusetts, October 1–4, 1996.

[33] Daniel J. Magenheimer, Liz Peters, Karl Pettis, and Dan Zuras. Integer multiplication and division on the hp precision architecture. In *Proc. Second International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS-II)*, pages 90–99, October 5–8, 1987.

[34] Tony Alan Marshall, Igor Kostarnov Stansfield, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proc. International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pages 135–143, Monterey, California, February 21-23, 1999.

[35] Motorola, Inc. *AltiVec Technology Programming Environments Manual*, November 1998. Available as: `http://www.motorola.com/SPS/PowerPC/teksupport/teklibrary/manuals/altiv%ec_pem.pdf`.

[36] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, January 1997.

[37] J. A. Perkins. Programming practices: Analysis of Ada source developed for the air force, army, and navy. In *Conference Proceedings on Ada Technology in Context: Application, Development, and Deployment (TRI-Ada '89)*, pages 342–354, Pittsburgh, Pennsylvania, October 22–26, 1989.

[38] A. Rashid, J. Leonard, and W.H. Mangione-Smith. Dynamic circuit generation for solving specific problem instances of boolean satisfiability. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 196–204, Napa Valley, California, April 15–17, 1998.

[39] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. 27th International*

Symposium on Microarchitecture (MICRO-27), pages 172–180, San Jose, California, November 1994.

[40] Jeff Reilly. SPEC describes SPEC95 products and benchmarks. *SPEC Newsletter*, September 1995. Available as: `http://www.spec.org/osg/cpu95/news/cpu95descr.html`.

[41] Stephen E. Richardson. Exploiting trivial and redundant computation. In *Proc. Eleventh Symposium on Computer Arithmetic*, pages 220–227, Windsor, Ontario, June 29–July 2, 1993.

[42] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *Proc. 30th International Symposium on Microarchitecture (MICRO-30)*, pages 248–258, Research Triangle Park, North Carolina, December 1–3, 1997.

[43] Jonatahan L. Schilling. Dynamically-valued constants: An underused language feature. *ACM SIGNPLAN Notices*, 30(4):13–20, April 1995.

[44] Michael D. Smith. Extending SUIF for machine-dependent optimizations. In *Proc. First SUIF Compiler Workshop*, pages 14–25, Stanford, California, January11–13, 1996. Stanford University.

[45] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *Proc. 30th International Symposium on Microarchitecture (MICRO-30)*, pages 281–290, Research Triangle Park, North Carolina, December 1–3, 1997.

[46] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[47] P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Accelerating boolean satisfiability with configurable hardware. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 186–195, Napa Valley, California, April 15–17, 1998.

Appendix A

SUIF Optimizations

This appendix describes compiler optimizations that were applied to each program before profiling. All optimizations are standard options in SUIF1 and their descriptions are copied directly from the `porky` man page. Successfully applied optimizations include:

Forward propagation

(`-forward-prop`) This forward propagates the calculation of local variables into uses of those variables when possible. The idea is to give more information about loop bounds and array indexing for doing dependence analysis and loop transformations, or generally to any pass doing analysis.

Constant propagation

(`-const-prop`) This does simple constant propagation.

Dead code elimination

(`-dead-code`) This does simple dead-code elimination.

Variable privatization

(`-privatize`) This privatizes all variables listed in the annotation "privatizable" on each TREE_FOR.

(`-glob-priv`) Do some code transformations to help with privatization of global variables across calls. It looks for "possible global privatizable" annotations on proc_syms. In each such annotation it expects to find a list of global variables. It changes the code so that a new parameter is added to the procedure for each symbol in the annotation, and all uses of the symbol are replaced by indirect references through the new parameter, and at callsites the location of that symbol is passed. If the procedure is a Fortran procedure, the new parameter is a call-by-ref parameter. It arranges for this to work through arbitrary call graphs of procedures. The result is code that has the same semantics but in which the globals listed in each of these annotations are never referenced directly, but instead a location to use is passed as a parameter. If the annotations are put on the input code properly, this allows privatization of global variables to be done as if the globals were local.

Common subexpression elimination

(`-cse`) This does simple common sub-expression elimination.

Hoist loop invariants

(`-loop-invariants`) This moves the calculation of loop-invariant expressions outside loop bodies.

(`-loop-cond`) Move all loop-invariant conditionals that are inside a TREE_LOOP or TREE_FOR outside the outermost loop.

Iterate optimizations

(`-iterate`) This flag says to keep doing all the specified optimizations as long as any of them make progress.

Additional, useful optimizations that were not successfully applied due to software bugs include:

Constant folding

(`-fold`) This folds constants wherever possible.

Reduction

(`-reduction`) This finds simple instances of reduction. It moves the summation out of the loop.

(`-ivar`) This does simple induction variable detection. It replaces the uses of the induction variable within the loop by expressions of the loop index and moves the incrementing of the induction variable outside the loop.

Appendix B

The Compilation Sequence

Table B.1 below describes the sequence of steps for compiling a self-profiling application. These steps include preprocessing, a number of SUIF passes, instrumentation, and subsequent emission and recompilation of C. Subsequent steps communicate through intermediate files, as listed in the **Inputs** and **Outputs** columns. Some of the steps can work incrementally on one source file at a time. However, many of the SUIF-based transformations (notably `linksuif`, `dup_procs`, and `instrument`) must process an entire file set at once. Rules for the entire compilation sequence are programmed into a Makefile that is included by the custom Makefile of each application.

Command	Inputs	Outputs	Description
`perl prof_prep`	`*.c`	`*.c`	Prepend: `#include <prof_runtime.h>` (profiling library header) into each C source file
`scc`† `-.spd`	`*.c`	`*.spd`	SUIF front end (preprocess and read C into internal form)
`linksuif`†	`*.spd`	`*-l.spd`	SUIF linker (reconcile global symbols in internal form)
`porky`† `-globalize`	`*-l.spd`	`*-g.spd`	Convert `static` local variables into globals
`dup_procs`	`*-g.spd`	`*-c.spd`	Duplicate procedures for call chain disambiguation (optional)
`porky`† `-forward-prop -const-prop -dead-code -glob-priv -privatize -cse -loop-invariants -loop-cond -iterate`	`*-c.spd`	`*-o.spd`	Compiler optimizations
`porky`† `-Dfors`	`*-o.spd`	`*-d.spd`	Dismantle `for` loops into `do...while` form
`instrument`	`*-d.spd`	`*-i.spd`	Instrument for profiling
`s2c`†	`*-i.spd`	`*-i.c`	Convert to C
`gcc -c`	`*-i.c`	`*-i.o`	Compile to object code
`gcc -lprof_runtime`	`*-i.o`		Link executable with profiling library `prof_runtime.a`

†Part of the SUIF distribution

Table B.1: Sequence of steps for compiling a self-profiling application