Detecting and Understanding Dynamically Dead Instructions for Contemporary Architectures

Marianne J. Jantz

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Committee:

Dr. Prasad Kulkarni: Chairperson

Dr. Xin Fu

Dr. Man Kong

Date Defended

The Thesis Committee for Marianne J. Jantz certifies
That this is the approved version of the following thesis:

Detecting and Understanding Dynamically Dead Instructions for Contemporary Architectures

Committee:

____
Chairperson

____

____

____
Date Approved

Acknowledgements

I would like to thank my advisor, Professor Prasad Kulkarni, for mentoring me during my research on this project. He often went above and beyond his responsibilities as a professor and was extremely patient with me. Wihtout his guidance, none of this would have been possible.

I would like to thank all the other professors who have taught and advised me during my time at the University of Kansas. I would like to thank Professor Nancy Kinnersley, who advised me during my undergraduate schooling and Professor Man Kong, who encouraged me to continue with graduate school. I would also like to thank Professors Perry Alexander, Andy Gill, and Xin Fu, who, along with my labmates, have made me feel welcome in the Computer Systems Design Laboratory.

I would like to thank the ITTC help team (especially Chuck Henry) for their excellent system administration and for introducing me to the cluster here at ITTC. Without this computing power, I would not have been able to complete many of my experiments.

I would like to thank my parents and sisters for all of their love and support. Finally, I would especially like to thank my brother and labmate who was always there to offer advice, which I sometimes took, and listen to my concerns, whether he was willing or not. This means the world to me.

Thank you all.

Abstract

Instructions executed by the processor are dynamically dead if the values they produce are not used by the program. Executing such *useless* instructions can potentially slow-down program execution and waste power. The goal of this work is to quantify and understand the occurrence of dynamically dead instructions (DDI) *for programs compiled using modern compilers for the most relevant contemporary architectures*. We expect our extensive study to highlight the issue of DDI and to play a critical role in the development of compiler and/or architectural techniques to avoid DDI execution at runtime.

In this thesis, we introduce our novel GCC-based instrumentation and analysis framework to determine DDI during program execution. We present the ratio and characteristics of DDI in our benchmark programs. We find that programs compiled with GCC (with and without optimizations) execute a significant fraction of DDI on x86 and ARM based machines. Additionally, an ample amount of predication employed by GCC results in a large fraction of executed instructions on the ARM to be dynamically dead. We observe that a handful of static instructions contribute a large majority to the overall DDI in standard benchmark programs. We also find that employing a small amount of static *context* information can significantly benefit the detection of DDI at run-time. Additionally, we describe the results of our manual study to analyze and categorize the DDI instances in our x86 benchmarks. We briefly outline compiler and architecture based techniques that can be used to eliminate each category of DDI in future programs. Overall, we believe that a close synergy between compiler and architecture techniques may be the most effective strategy to eliminate DDI to improve sequential program performance and energy efficiency on modern machines.

Contents

Abstract **iii**

Table of Contents **iv**

List of Figures **vi**

1 Introduction **1**

2 Related Work **5**

3 Framework for Exploring Dynamically Dead Instructions **11**
3.1 Generating Dynamic Program Profile 12
3.2 Finding Dynamically Dead Instructions 13
3.3 Benchmark Set and Simpoints 16

4 Experimental Results and Analysis **18**
4.1 x86 Results and Analysis 18
4.1.1 Ratio of Dynamically Dead Instructions 19
4.1.2 Context Information to Improve DDI Detection 28
4.1.3 Understanding and Characterizing Dynamically Dead Instructions 33
4.2 ARM Results and Analysis 38
4.2.1 Implementation and Challenges 38
4.2.2 Ratio of Dynamically Dead Instructions 44
4.2.3 Static Instructions and Dynamically Dead Instructions . . 46

5 Future Work **49**

6 Conclusions **50**

References **52**

List of Figures

2.1 Varieties of dead code elimination optimizations in a compiler . . 6

3.1 Sample example to illustrate the backward traversal algorithm to dead dynamically dead instructions 14

4.1 Percentage of dynamically dead instructions in x86 benchmark programs. The DDI is further categorized as register set dead instructions, memory set dead instructions, and *NOP* instructions 19
4.2 Optimization to reduce load/store latency (dijkstra) 20
4.3 Number of *static* instruction instances corresponding to DDI for optimized x86 MiBench benchmarks. The three bars for each benchmark display static instructions reached without context information, with a single *callee-static PC* function for context information, and using the entire *callee-static PC* function stack as context respectively. Note, the vertical axis is plotted on a logarithmic scale 22
4.4 Number of *static* instruction instances corresponding to DDI for optimized x86 SPEC benchmarks. The three bars for each benchmark display static instructions reached without context information, with a single *callee-static PC* function for context information, and using the entire *callee-static PC* function stack as context respectively. Note, the vertical axis is plotted on a logarithmic scale 22

4.5 Number of *static* instruction instances corresponding to DDI for unoptimized x86 MiBench benchmarks. The three bars for each benchmark display static instructions reached without context information, with a single *callee-static PC* function for context information, and using the entire *callee-static PC* function stack as context respectively. Note, the vertical axis is plotted on a logarithmic scale . . . 22
4.6 Number of *static* instruction instances corresponding to DDI for unoptimized x86 SPEC benchmarks. The three bars for each benchmark display static instructions reached without context information, with a single *callee-static PC* function for context information, and using the entire *callee-static PC* function stack as context respectively. Note, the vertical axis is plotted on a logarithmic scale 23
4.7 Contributions of static (partially) dead instruction instances to the DDI of optimized x86 MiBench benchmarks (sorted in ascending order) . . . 24
4.8 Contributions of static (partially) dead instruction instances to the DDI of optimized x86 SPEC benchmarks (sorted in ascending order) 24
4.9 Contributions of static (partially) dead instruction instances to the DDI of unoptimized x86 MiBench benchmarks (sorted in ascending order) . . . 24
4.10 Contributions of static (partially) dead instruction instances to the DDI of unoptimized x86 SPEC benchmarks (sorted in ascending order) . . . 25
4.11 Percentage of DDI that are dead with 100% probability in the optimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information . . . 25
4.12 Percentage of DDI that are dead with 90% probability in the optimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information . . . 26

4.13 Percentage of DDI that are dead with 70% probability in the optimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information 26
4.14 Percentage of DDI that are dead with 100% probability in the unoptimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information 28
4.15 Percentage of DDI that are dead with 90% probability in the unoptimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information 28
4.16 Percentage of DDI that are dead with 70% probability in the unoptimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information 29
4.17 Preliminary analysis of the occurrence of dynamic dead instructions 34
4.18 Relative categories of dead instructions in each benchmark. The left and right bars for each benchmark show DDI in unoptimized and optimized codes respectively. 37
4.19 Example RTLs generated for the optimized ARM *ispell* benchmark with corresponding assembly instructions 40
4.20 Percentage of dynamically dead instructions in ARM benchmark programs 44
4.21 Percentage of dynamically dead instructions in ARM benchmark programs categorized as register set dead instructions, memory set dead instructions, predicated dead instructions, and *NOP* instructions 45
4.22 Number of instruction instances reached over execution of ARM MiBench benchmarks 46

4.23 Contributions of static (partially) dead instruction instances to the DDI of optimized ARM MiBench benchmarks (sorted in ascending order) 47
4.24 Contributions of static (partially) dead instruction instances to the DDI of unoptimized ARM MiBench benchmarks (sorted in ascending order) 47

Chapter 1

Introduction

Physical barriers and technology limitations have effectively ended the era of rapid processor frequency scaling to automatically increase *single-thread* software performance. However, even as the migration of computer hardware into the multi/many-core domain is now several years old, we find that many software tasks are inherently sequential and derive no benefit from increasing processor counts on multi-core machines. Additionally, even most parallel workloads are limited by their sequential counterparts, as dictated by Amdahl's law, and do not scale beyond a small number of cores. Therefore, we believe that it will become attractive as well as important in the future to dedicate more transistors to improve single-thread performance. Similarly, aggressive techniques are necessary to improve single-thread program performance on modern machines. In this research we investigate the phenomenon of *dynamically dead instructions* (DDI) and their potential to benefit sequential program performance and energy efficiency.

Researchers have observed that a surprisingly large fraction of the instructions executed by a processor are often *dead*, that is, their calculated result is not used by the program [2]. It was reported that, on average, close to 20% of the instructions

executed by programs can be dead (even excluding NOP instructions) [4, 17]. It is obvious that executing dynamically dead instructions will waste power and hardware resources, and likely slow-down the program execution.

The issue of DDI is well-known, old, and fundamental. However, we believe that this issue may have received less attention during the earlier era of exponentially growing uniprocessor clock speeds, when single-threaded applications were enjoying free, regular, and rapid performance gains, and microprocessor energy consumption was not as important of an issue. With physical barriers and technology limitations causing a stagnation of single-core program performance, techniques to achieve automatic efficiency improvements for all existing and future microprocessors are becoming very critical. Similarly, mechanisms to reduce power consumption are also important to improve the operational characteristics of embedded and battery-operated devices, as well as, large server farms. Eliminating DDI will automatically achieve the efficiency and power benefits for all program threads, and thus satisfy both these major computing trends.

All previous DDI studies were conducted on architectures such as the Alpha and the Itanium that are defunct or less mainstream today. This work revisits and investigates the issue of DDI for more contemporary and relevant architectures, such as the x86 and the ARM. Similarly, this work also seeks to gain a comprehensive understanding of the characteristics of DDI in existing programs to be able to develop new software and hardware-based DDI elimination techniques. We also investigate the types and properties of DDI, and systematically characterize them for programs compiled using modern, state-of-art compilers.

In this paper, we describe the framework we built to detect, study, and categorize the DDI in x86 and ARM benchmark programs. Our detection framework uses

GCC to instrument the binaries with additional instructions to produce control-flow and data-flow traces on program execution. We implement algorithms to analyze the dynamic trace to determine the number and ratio of DDI, and their corresponding static instructions. We also determine the ratio and number of static instructions that actually contribute to the overall DDI. We further explore the effect of using static *context* information to isolate instructions with a high probability of being dynamically dead. We also manually analyzed the DDI in our smaller benchmark programs to better understand the causes for dead instructions. We use this analysis to categorize the DDI and propose static compiler approaches to eliminate DDI from binary programs, and (micro) architectural techniques that employ compiler-driven feedback to avoid the execution of DDI at runtime. Thus, the major contributions of this work are the following:

1. This is the first work to quantify and study the properties of DDI for contemporary architectures, the x86 and the ARM, compiled and optimized with a modern compiler,
2. This is the first investigation into the probability that an instruction will be dynamically dead, as well as the effect of using static context information to find highly probable dead instructions, and
3. This work presents our observed categorization of DDI for optimized and unoptimized versions of x86 programs.

The rest of this thesis is organized as follows. We present background concepts and related work in the area of dynamically dead instruction detection and elimination in Chapter 2. We describe our GCC-based framework to detect, analyze, and categorize DDI in Chapter 3. We present our experimental results and

observations in Chapter 4. Finally, we describe our plans for future work and the conclusions from this research in Chapters 5 and 6 respectively.

Chapter 2

Related Work

In this section, we describe background concepts and related work in the areas of characterizing dead instructions and compiler and architectural techniques for eliminating them. Earlier research efforts have explored both software and hardware approaches to address the problem of DDI. Unreachable and dead code can be introduced by software developers into high-level language programs or by the compiler while optimizing and generating binary code. Traditional compiler optimizations, such as *unreachable code elimination*, *dead code elimination,* and *partial dead code elimination* are tasked with detecting and removing such dead code from generated programs. While, *unreachable* and *dead code elimination* detect and remove code that is dead along all program paths from the program start, *partial dead code elimination* is a more complex algorithm that attempts to find code that is useful on some program paths, while being dead on the other paths.

Figure 2.1 presents examples to illustrate *fully dead* and *partially dead* code in programs. Figure 2.1(a) shows an instance of full dead code elimination, where the assignment to `y` in block #1 is never used before being reset in block #5 along

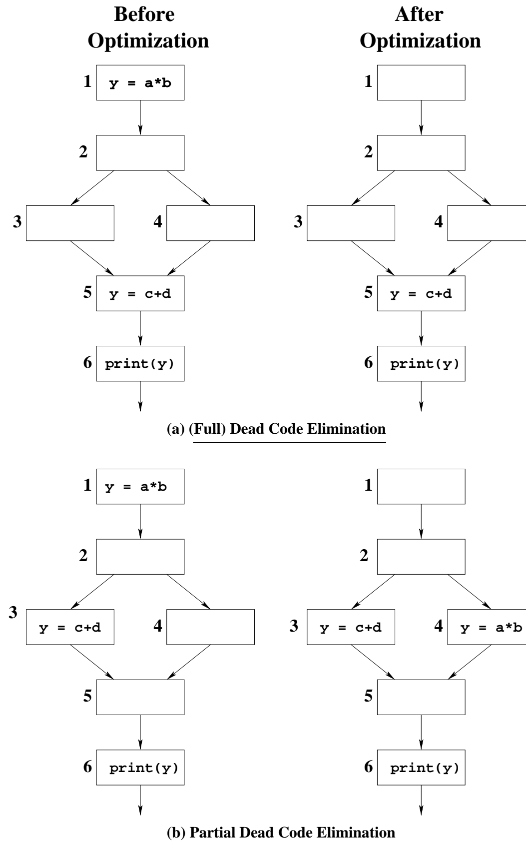


Figure 2.1. Varieties of dead code elimination optimizations in a compiler

all program paths. The compiler removes such dead assignments from optimized codes. Figure 2.1(b) shows an example of *partial dead code elimination*. In contrast to the previous example where the dead statement was reset before being used along all program paths, the assignment to `y` in block #1 of Figure 2.1(b) is reset (in block #3) before being used along the program path 1-2-3-5-6, but is used along the other path 1-2-4-5-6. The compiler optimization algorithm of *partial dead code elimination* can handle such code by aggressively moving the partially dead statements down in the control-flow as far as possible, while maintaining the program semantics [7]. The second graph in Figure 2.1(b) illustrates the resulting code after applying this optimization. Although these optimizations are highly

effective in removing many dead instructions from generated codes, high rates of DDI still persist even for programs generated by modern sophisticated compilers that contain and apply these optimizations. Our research attempts to investigate why these optimizations, as implemented in existing compilers, are not always effective at eliminating dead instructions in the program, and its repercussions on performance.

Butts and Sohi proposed a mechanism for the microprocessor hardware to predict and eliminate dynamically dead instructions at runtime [2]. This work only tracked instructions that produce dead register values (and ignored dead memory stores, nops and prefetches) to simplify their detection and remedial mechanisms. Even with this restriction, they observed that between 3% to 16% of the instructions executed by the SPEC2000 integer benchmarks using the Alpha instruction set were dead. They also noticed that many dead instructions are introduced by the compiler during code optimizations, like *instruction scheduling*. They developed a hardware unit to predict dead instructions in the dynamic instruction stream. Their predictor achieved good accuracy and, along with some other cache-based hardware, was able to avoid the execution of 79% of useless instructions in their benchmarks. This DDI elimination achieved up to 9.6% speedup benefits. However, this study did not perform a thorough investigation and categorization of dead instructions across different compilers and architectures, especially for those that are more prevalent today. This was also a pure hardware study and did not propose any compiler techniques to eliminate DDI, evaluate their costs, and study interactions with other compiler optimizations.

Related also is the work of Sundaramoorthy et al. that proposed a new processor microarchitecture that simultaneously runs two copies of every program to

exploit the properties of *predictable* dead, branch, and other *ineffectual* instructions to speed up both the duplicated program streams [23]. In their scheme, the first *speculative* thread runs faster by skipping over instructions whose results in their previous instances were predicted correctly, and uses their predicted values instead. The other thread that validates these predictions can also speed up since it has a more accurate picture of the future. Thus, the two redundant program threads combined run faster than either can alone. This work also did not attempt to investigate the causes or devise techniques to eliminate DDI in code generated by the compiler, and is very resource intensive for routine deployment in all processors. At the same time, attempts to address the DDI issue with architectural and/or microarchitectual changes have not been adopted, likely due to high associated design and implementation costs. However, modern device technologies may make it possible to develop other more successful hardware schemes to handle DDI at run-time.

Researchers have also explored static instructions that produce the same value on multiple consecutive dynamic invocations [10], or those that update a register or memory location with a value that it already contains [9]. This phenomenon is called value locality. Related research attempts to detect dynamic instructions that update a register or memory location with a value that it already contains [9, 14]. Some researchers have explored the phenomenon of *silent stores*, which are memory write instructions that do not alter the value already present at the target address [9, 26]. Many of these works also propose and evaluate speculative mechanisms to remove or eliminate such *ineffectual* instructions. We do not consider such categories of instructions, since they are not statically dead from the compiler's point of view.

Some other works have observed and exploited the occurrence of dynamically dead instructions in executed programs. Lumetta and Patel found that, on average, 15% of all dynamically executed instructions in SPEC2000 integer benchmarks on the Alpha processor are dead [11]. They also measured an additional 10% of the instructions to be *nops*. Fahs et al. proposed the *rePLay* microarchitecture to provide dynamic optimization support at the microarchitecture level. Their dynamic optimization system built upon the Alpha simulator discovered 24% of dynamic instructions to be dead, on average, and eliminated about 10% of them. Again, none of these approaches investigate the causes of DDI, study this phenomenon for contemporary compilers and architectures, or suggest mechanisms to reduce or eliminate them from binary programs.

Detecting and understanding dynamically dead instructions will require us to generate and analyze the profile or trace of the whole program execution. Compiler and computer architecture researchers have often employed such execution time program trace information to understand important program properties [1, 16, 19]. The first algorithms for generating whole-program paths were presented by Larus [8] and Melski and Reps [12]. These algorithms instrument the program to generate a complete trace of all *basic blocks* or paths executed by the program. Later, researchers extended these algorithms so that the instrumented programs also generate the memory dependence profile of the program, which is necessary to detect dead memory load instructions [25, 28]. While the naive generation of whole program traces is relatively simple, the collected traces are often extremely long. Consequently, most research is focused on developing compression algorithms to compact the larger generated traces [18, 27]. We will use and extend these algorithms to generate the control-flow and data-flow profiles

for this research.

Chapter 3

Framework for Exploring Dynamically Dead Instructions

In this section we will describe our framework to generate program execution profiles to detect and investigate dynamically dead instructions. We employ and modify the GCC compiler (version 4.5.2) for this research. We use the same GCC source code to build an ARM cross-compiler toolchain (cross-platform arm-elf-gcc compiler, cross-platform binutils). The program is instrumented after all the optimizations are applied and immediately before code generation. The binaries are generated for a 32-bit x86 platform, as well as the ARM platform. Each x86 binary is natively executed on a server machine with Intel(R) Xeon(R) processors. Each ARM binary is natively executed on an OMAP4 Panda board with a dual-core ARMv7 Processor. The instrumented binaries generate trace files, which we later use to analyze and discover instances of DDI in our benchmark programs. In this section, we provide further details on our compiler-based implementations.

3.1 Generating Dynamic Program Profile

Dynamic program profiles can be straightforwardly produced in one of two ways: (a) by modifying the compiler to instrument the generated binary with additional instructions to output some representation of the trace when the program is run, or (b) by updating a processor simulator/emulator to output the instructions that it executes for an unmodified binary. We choose to use option (a) for our trace generation. We believe that the mechanism of generating traces via compiler-inserted instrumentations has advantages over the simulator-based approach:

1. A compiler-based mechanism will be more flexible, for example, by easily allowing the selective instrumentation of only application functions, or all application and library functions,
2. The compiler-based approach can allow the instrumented binaries to run natively on available architectures, including x86 and ARM, which is much faster than using a simulator, and
3. The backward-scan algorithm to analyze the dynamic trace and detect DDI needs to parse each instruction to determine the registers or memory locations that are set or used. Thus, to study the issue of DDI across multiple architectures, a simulator-based approach may require us to implement this algorithm over architecture-specific assembly instructions, which will necessitate understanding the instruction format and updating the implementation for each architecture. However, many compilers use a common low-level intermediate language (IR), like the RTLs used by GCC, that has a one-to-one correspondence with assembly instructions. Such correspondence will

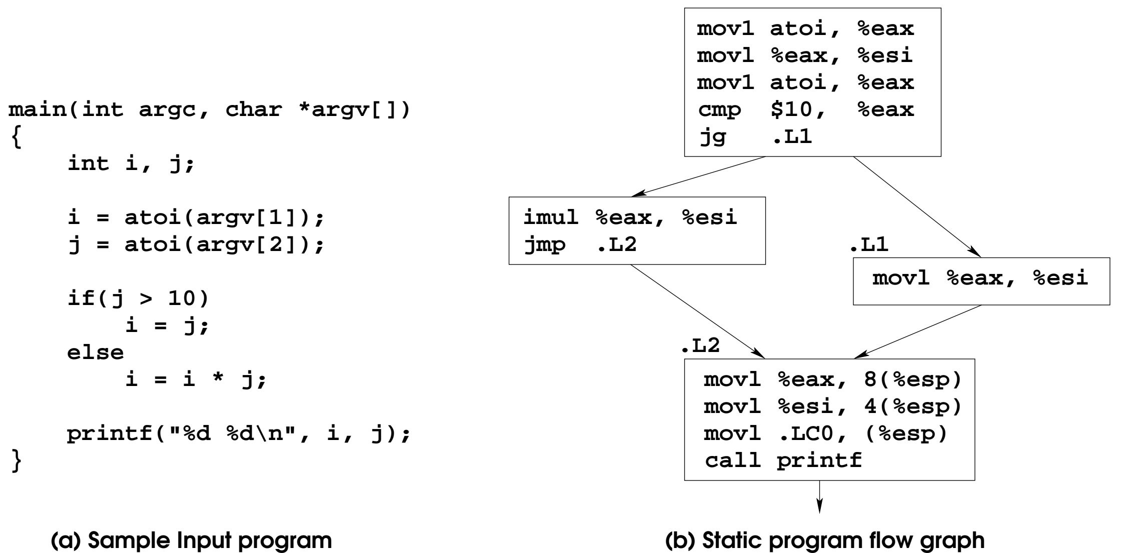
allow us to implement our parsing algorithm only once for the compiler IR, and not update it for each architecture.

The compiler-based method does have one drawback. While the compiler can easily instrument application and library functions, this approach may find it more difficult to instrument system calls and trap routines. However, even most simulators only *emulate* system calls, which implies that the compiler-based approach will not result in reduced trace accuracy in such cases. In future work, we will evaluate the potential benefit to accuracy of instrumenting system calls as compared to the alternative of conservatively assuming that all/most program register and memory state is used on entering system functions.

For this work, we modify the GCC compiler to insert instrumentation code into *only* the application binaries during its final code generation pass, after all the optimization phases have been applied. We do not yet instrument library functions. Our tracing algorithm automatically marks all arguments passed to a library function as being used. The compiler also produces a new file containing a numbered list of all the basic blocks (along with their constituent instructions) in the program. The inserted instrumentations produce two trace files on program execution. One trace file contains an uncompressed sequential list of the basic block numbers as they are reached during execution. The other file contains a list of memory addresses as they are accessed during execution.

3.2 Finding Dynamically Dead Instructions

The dynamic program execution trace in its most basic form consists of a linear sequence of instructions (or basic blocks) in the order they are executed by the processor. Therefore, algorithms for finding dynamically dead instructions



`movl atoi, %eax`	Set:	Used:	*; t1 = atoi(argv[1]), transitively dead*
`movl %eax, %esi`	Set:	Used:	*; i = t1 , dead assignment to i*
`movl atoi, %eax`	Set: %eax	Used: %eax	*; j = atoi(argv[2])*
`cmp $10, %eax`	Set:	Used: %eax	*; compare j with 10*
`jg .L1`	Set:	Used: %eax	*; if(j>10) jump to block L1*
`movl %eax, %esi`	Set: %esi	Used: %eax, %esi	*; i = j*
`movl %eax, 8(%esp)`	Set:	Used: %eax, %esi	*; push j on the stack for printf*
`movl %esi, 4(%esp)`	Set:	Used: %esi	*; push i on the stack for printf*
`movl .LC0, (%esp)`	Set:	Used:	*; push printf string on the stack*
`call printf`	Set:	Used:	*; printf uses i and j*

(c) Dynamic program trace (./a.out 4 20) and analysis to detect dynamically dead instructions

Figure 3.1. Sample example to illustrate the backward traversal algorithm to dead dynamically dead instructions

in program execution traces only need to perform a single sequential scan of the trace. Most algorithms scan the trace in reverse order to reduce the complexity of classifying dead instructions. In particular, when processing a particular instruction in the trace, reverse scanning allows the liveness value of all consumers of the instruction's result to be already known [2, 4].

We use a simple example program in Figure 3.1 to illustrate the typical process of generating the dynamic program trace and analyzing it for dynamically dead instructions. Figure 3.1(a) shows the example 'C' program that initializes local

variables `i` and `j` with input arguments entered on the command-line. While the initialized value of `j` is used along both paths of the *if*-branch, `i` is only used along one path. Thus, the initialization of `i` in the statement `i=atoi(argv[1])` is partially dead. Figure 3.1(b) shows the static control-flow graph of the code generated by GCC for the example C program for x86 32-bit architecture. To keep this example simple, we have left out the code generated by the compiler for managing the run-time stack, and abstracted the calls to *atoi()* with the two `movl atoi, %eax` instructions to consecutively initialize `i` and `j` respectively. Thus, we can see that the compiler did not eliminate the partially dead assignment to `i` (`movl %eax, %esi`) from the generated optimized binary code.

Figure 3.1(c) represents the dynamic trace that is generated on executing the binary program in Figure 3.1(b) with the input arguments *4 and 20* respectively (i=4 and j=20). It is important to note that the dynamic program trace is a linear sequence of instructions with no control-flow transfers, which makes it easier to build algorithms to analyze the trace. The algorithm to detect dynamically dead instructions scans the trace in reverse order, starting at the last instruction. Figure 3.1(c) also shows the lists of *Set* and *Used* registers that can be maintained during this scan. (Again, to keep this example simple, we only track and show the register sets/uses, and ignore memory loads/stores.) Thus, during this backward scan, if we reach an instruction that sets a register or memory location when that register or address is not on the *Used* list, then we tag that instruction as dead. Therefore, the second instruction in the dynamic trace, `movl %eax, %esi`, will be tagged as a *direct* dead instruction. The registers/addresses used in such dead instructions will not be put on the *Used* list, since they do not produce useful values. Thus, the register `%eax` is not inserted in the *Used* list for this second

instruction in the trace. Consequently, the first trace instruction that sets `%eax` is also marked as a *transitively* dead instruction. We use this simple linear-time algorithm to detect the dynamically dead instructions in a single pass over the program execution trace.

3.3 Benchmark Set and Simpoints

Our benchmark set included programs from the MiBench [5] and the SPEC CPU2006 benchmark suites [3]. MiBench includes 'C' programs generally used in embedded applications. We randomly select one program from each of the six MiBench categories for our set. The standard SPEC CPU 2006 benchmarks contain larger CPU-intensive general-purpose applications. We include eight 'C' integer benchmarks from the SPEC CPU 2006 set in our experiments. While our x86 experiments use our complete benchmark set, we only use the embedded MiBench benchmarks on the ARM platform.

The *reference* input data set provided with the SPEC benchmarks results in very long-running programs. Therefore, as is commonly done in most architectural studies, we employ the Simpoint framework to limit the program run-times with SPEC benchmarks, and the corresponding size of the generated dynamic program traces, while still gathering enough information to perform accurate DDI analysis and characterization of the full program [20]. The Simpoint framework allows us to generate traces over smaller intervals of the benchmarks instead of the entire execution of each SPEC benchmark. With Simpoints, information was gathered for a maximum of five 100 million instruction windows for each SPEC benchmark.

In order to determine and find the representative program traces, the Simpoint tool requires us to perform off-line analysis to generate Basic Block Vectors (BBV)

for each benchmark, input, compiler configuration, and architecture combination. A BBV is a list of the number of times each basic block is entered during a program execution run. The Simpoint analysis tool used the BBV to find regions of code with similar execution behavior (program phases).

We again employed our compiler-based instrumentation methodology (mentioned in Section 3.1) to determine the BBV for each configuration. However, instead of generating a trace file of the basic blocks as they are reached during execution, we create our basic block vectors and dump them in 100 million instruction intervals to generate the BBVs. This BBV file is then fed to the Simpoint tool, which outputs the *simpoints* for each benchmark configuration, along with a *weight* for each simpoint. We use the weight to scale each phase interval to extrapolate the DDI behavior over the full-program.

One challenge when performing our DDI analysis over the generated simpoints is caused by the fact that the intervals usually start and end in the middle of the benchmark's execution. It may happen that a memory location *set* in the interval might not be *used* until after that interval. Thus, technically, the set of that memory location is not a dead instruction, but our scan algorithm finds it so. In such cases, we choose to keep our analysis conservative and consider all memory locations and registers as used upon starting the *backward* scan on the simpoints. Thus, due to using the Simpoint tool, our DDI numbers for SPEC benchmarks reflect a conservative estimate.

Chapter 4

Experimental Results and Analysis

We use our modified version of GCC to instrument the x86 and ARM benchmark programs. These instrumented programs produce instruction and data traces at runtime, which we then analyze for DDI. For each benchmark, we use the GCC optimizations flags to generate and analyze both the unoptimized (-O0) and optimized (-O3) binaries. In this section we present the results of our analysis regarding the ratio and characteristics of DDI for both x86 and ARM binary programs generated by GCC.

4.1 x86 Results and Analysis

We first describe our DDI analysis results on the x86 platform in this section. Results of our experiments on the ARM are presented in Section 4.2. We use the algorithm described in Section 3.2 to traverse the execution traces for each x86 benchmark (or benchmark's simpoints) and collect the number and characteristics

of each program's dynamically dead instructions.

4.1.1 Ratio of Dynamically Dead Instructions

Figure 4.1 shows the ratio of the number of total executed instructions for each benchmark that are dynamically dead. On average, our unoptimized MiBench benchmarks contain 4.62% of DDI, while the optimized MiBench benchmarks have a slightly higher DDI fraction (7.71%). The SPEC benchmarks exhibit even higher DDI. Thus, the unoptimized SPEC benchmarks contain 8.71% of dynamically dead instructions, while the optimized programs display a larger percentage at 10.12% of DDI, on average. This observation of optimized programs containing more dead instructions is consistent with earlier research conducted on the Alpha architecture [2]. Figure 4.1 further breaks-up the DDI into three categories: dead

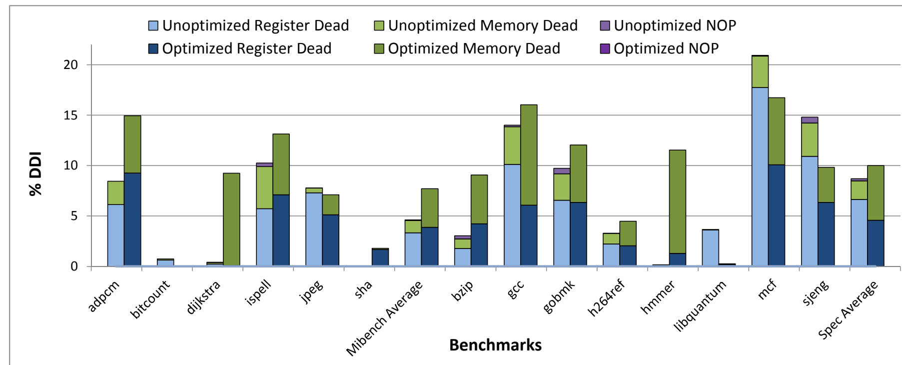


Figure 4.1. Percentage of dynamically dead instructions in x86 benchmark programs. The DDI is further categorized as register set dead instructions, memory set dead instructions, and *NOP* instructions

instructions due to a register set, dead instructions due to a memory set, and *NOP* instructions. A *NOP* instruction does not change the state of the system. It is used for several purposes such as to force memory alignment, to prevent hazards, etc. On average the unoptimized MiBench benchmarks contain 3.34% register set dead

instructions, 1.21% memory set dead instructions, and 0.06% *NOP* instructions. The optimized MiBench benchmarks contain 3.88% register set dead instructions, 3.83% memory set dead instructions, and no *NOP* instructions. Correspondingly, on average, the unoptimized SPEC programs contain 6.64% register set dead instructions, 1.85% memory set dead instructions, and 0.21% *NOP* instructions. The optimized SPEC benchmarks contain 4.58% register dead instructions, 5.42% memory set dead instructions, and again, no *NOP* instructions.

We can make several observations from this figure. First, all benchmarks contain both register and memory set DDI. Second, compiler optimizations are able to completely remove *NOP* instructions for x86 binaries. Third, all the benchmarks, with the exception of (optimized) bitcount, display a larger number of memory set dead instructions in the optimized binaries than in the unoptimized.

```
for (i=0 ; i<NUM_NODES ; i++) {
    if((iCost = AdjMatrix[iNode][i]) != NONE) {
        ...
    }
}
==> converted by compiler to
for (i=0 ; i<NUM_NODES ; i++ ) {
    leal  (%edi,%ebx), %ecx
    movl  AdjMatrix(,%ecx,4), %esi
    cmpl  $9999, %esi
    movl  %esi, iCost
    je    .L45 //if false, enter if statement
    ...
}
```

Figure 4.2. Optimization to reduce load/store latency (dijkstra)

There could be several reasons for this increase in the memory set DDI. For instance, we observed that GCC commonly performs an optimization that assigns a value to a register in order to reduce the load/store latency. Figure 4.2 shows an example of this optimization from the *dijkstra* benchmark. In this example, the unoptimized binary first sets the memory location of the variable `iCost`, and then checks whether the `if`-path is taken. Instead, the optimized binary stores the value of `iCost` both in memory and in the register `%esi`. Then, instead of using

the memory location holding the contents of `iCost`, the rest of the loop body uses the `%esi` register for the value of `iCost`. The `iCost` variable is updated once in each loop iteration. Thus, while this optimization will likely reduce the load/store latency, the set of the memory location has a very high probability of being dead. This specific example actually accounts for 48.56% of the DDI we discovered in the optimized x86 *dijkstra* benchmark.

4.1.1.1 Static Instruction Contributions to DDI

We have observed that most DDI are either partially dead or difficult to eliminate using pure static compiler analysis. Promising approaches to remove DDI will likely involve a hybrid compiler-hardware approach, where instructions tagged by the compiler as *probably dead* will be tracked by the hardware at run-time. For such techniques, success in eliminating DDI at low (hardware and power) costs will depend on the compiler accuracy of tagging potentially dead instructions, and the number of instructions that the hardware needs to track at run-time. In this section we collect statistics to determine the feasibility of such DDI elimination techniques.

The *first/leftmost bar* in Figures 4.3, 4.4, 4.5, and 4.6 show the number and ratio of *static* instruction instances that correspond to the DDI as compared to the total number of static instructions reached during the execution of each x86 benchmark. Thus, we can see that on average, for the optimized SPEC and MiBench benchmarks, about 18.62% and 8.97% of the static instruction instances contribute to the overall DDI respectively. These average ratios remain similar for the unoptimized SPEC and MiBench benchmarks, with about 18.25% and 7.64% of the static instructions contributing to the DDI respectively.

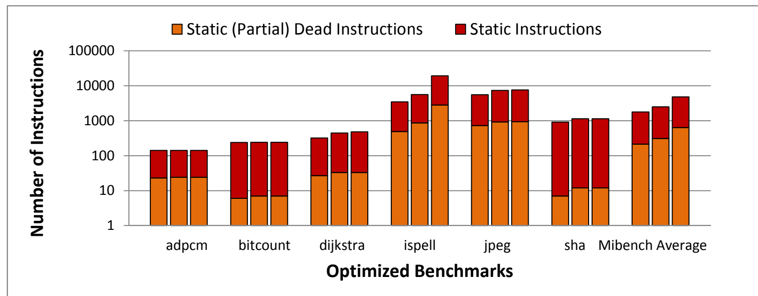


Figure 4.3. Number of *static* instruction instances corresponding to DDI for optimized x86 MiBench benchmarks. The three bars for each benchmark display static instructions reached without context information, with a single *callee-static PC* function for context information, and using the entire *callee-static PC* function stack as context respectively. Note, the vertical axis is plotted on a logarithmic scale

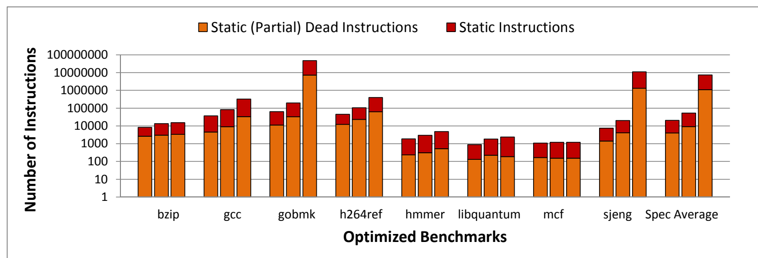


Figure 4.4. Number of *static* instruction instances corresponding to DDI for optimized x86 SPEC benchmarks. The three bars for each benchmark display static instructions reached without context information, with a single *callee-static PC* function for context information, and using the entire *callee-static PC* function stack as context respectively. Note, the vertical axis is plotted on a logarithmic scale

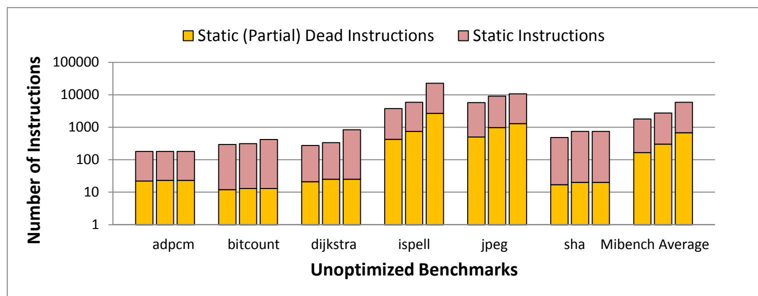


Figure 4.5. Number of *static* instruction instances corresponding to DDI for unoptimized x86 MiBench benchmarks. The three bars for each benchmark display static instructions reached without context information, with a single *callee-static PC* function for context information, and using the entire *callee-static PC* function stack as context respectively. Note, the vertical axis is plotted on a logarithmic scale

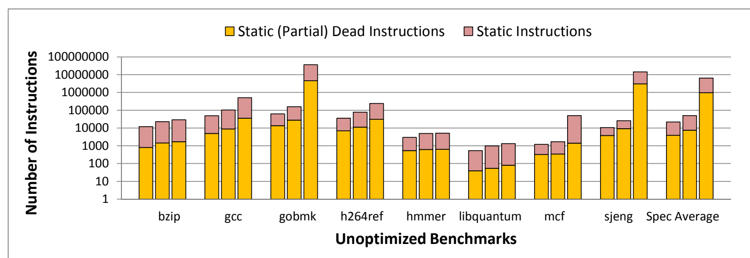


Figure 4.6. Number of *static* instruction instances corresponding to DDI for unoptimized x86 SPEC benchmarks. The three bars for each benchmark display static instructions reached without context information, with a single *callee-static PC* function for context information, and using the entire *callee-static PC* function stack as context respectively. Note, the vertical axis is plotted on a logarithmic scale

Figures 4.7, 4.8, 4.9, and 4.10 are plotted to further study *only* the set of static instructions that contribute to the program DDI. These figures sort (in ascending order) and display the contributions of individual static (partially) dead instruction instances to the overall percentage of dynamically dead instructions for the x86 benchmarks. Thus, we can observe an interesting pattern in these figures: a very small percentage of instruction instances actually contribute to the total DDI in our x86 benchmarks. For most of the benchmarks, whether they are unoptimized or optimized, about 80 to 90% of the static (partially) dead instructions do not contribute to the overall DDI in the SPEC benchmarks. The MiBench programs are a little more varied and contain fewer static instructions in general. Still, we can observe that about 70% of the static (partially) dead instructions do not contribute to 50% or more of the overall DDI for most of the benchmarks.

Potential techniques to detect and eliminate dynamically dead instructions will likely also be impacted by the *probability* of static instructions being dead at run-time. Probability for our purposes implies the ratio of the number of times that a static instruction is dead to the number of times it is encountered during

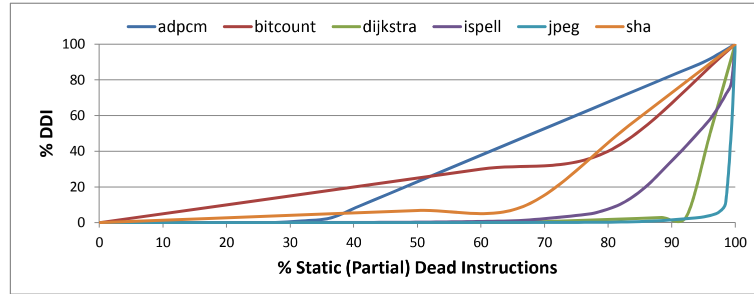


Figure 4.7. Contributions of static (partially) dead instruction instances to the DDI of optimized x86 MiBench benchmarks (sorted in ascending order)

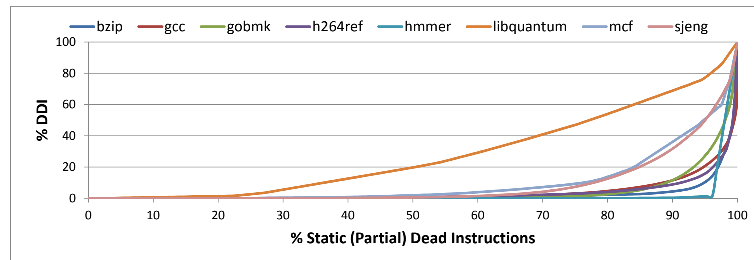


Figure 4.8. Contributions of static (partially) dead instruction instances to the DDI of optimized x86 SPEC benchmarks (sorted in ascending order)

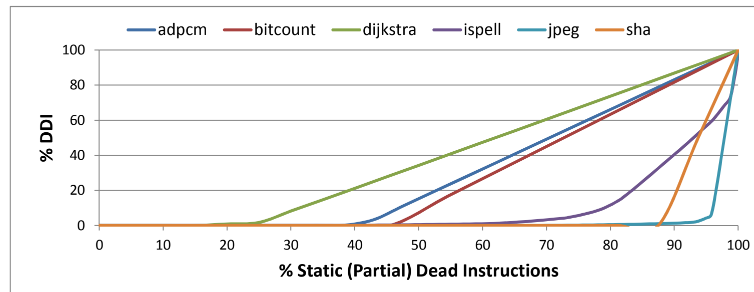


Figure 4.9. Contributions of static (partially) dead instruction instances to the DDI of unoptimized x86 MiBench benchmarks (sorted in ascending order)

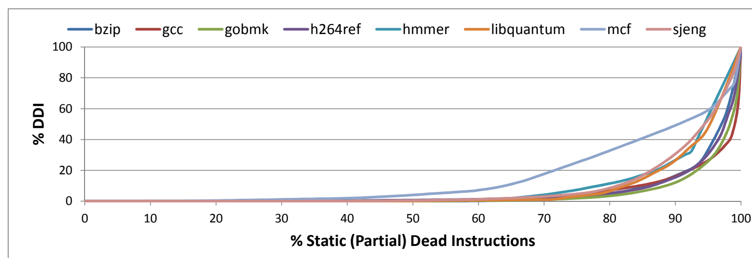


Figure 4.10. Contributions of static (partially) dead instruction instances to the DDI of unoptimized x86 SPEC benchmarks (sorted in ascending order)

program execution. Thus, if a specific static instruction is always dead, then we say that its DDI probability is 100%.

The leftmost bar for each benchmark in Figure 4.11 shows the percentage DDI that are dead with 100% probability in the optimized x86 benchmarks. Thus, on average, 0.14% and 1.73% of dynamically executed instructions are dead every time for the MiBench and SPEC benchmarks respectively. In other words, only about 2% and 17% of the DDI are dead with a 100% probability for our optimized MiBench and SPEC benchmarks respectively.

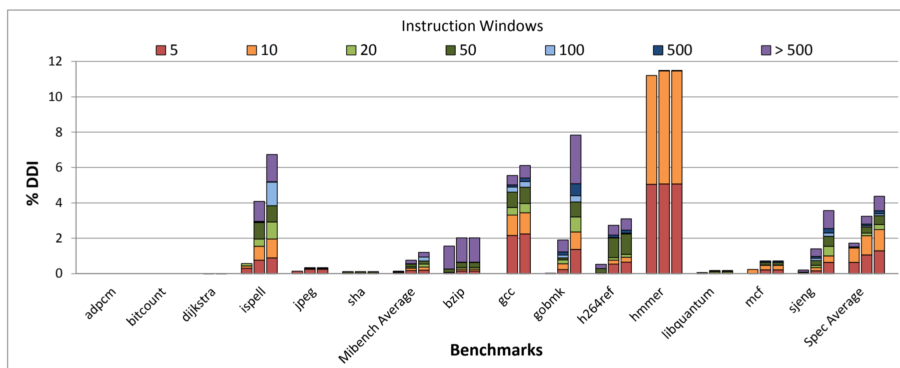


Figure 4.11. Percentage of DDI that are dead with 100% probability in the optimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information

We further categorize these results based on how fast (in terms of number of

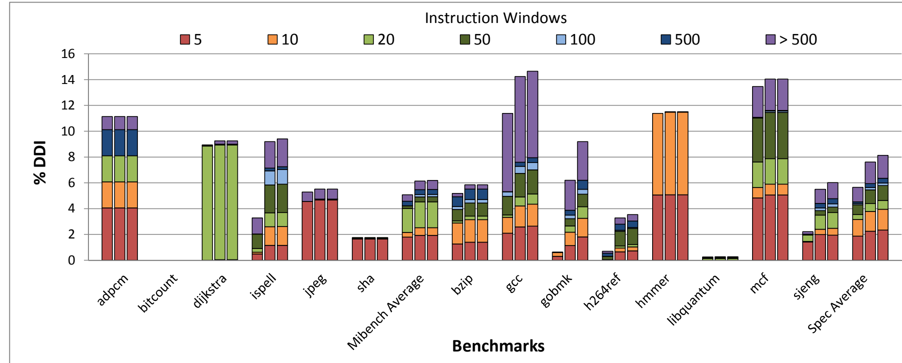


Figure 4.12. Percentage of DDI that are dead with 90% probability in the optimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information

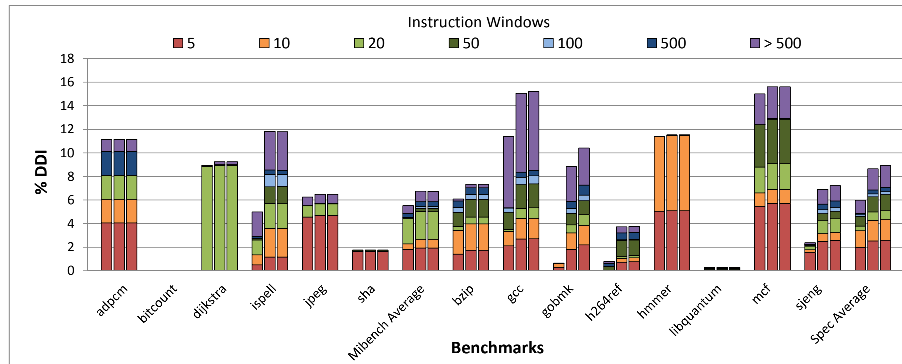


Figure 4.13. Percentage of DDI that are dead with 70% probability in the optimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information

intervening executed instructions) an instruction is detected to be dead after it is reached. This detection speed may affect how long a potential dead instruction may need to be delayed to avoid its execution for DDI elimination techniques and/or to develop other better and more informed online DDI elimination techniques. We further extended our trace algorithm to not only trace the register or memory location being set, but to additionally determine *when* the register

or memory location was (re)set. With this modification, we could also find dead instructions within specific instruction windows. We employ instruction windows of 5, 10, 20, 50, 100, 500, and >500 to analyze the speed of detecting dynamically dead instructions. Thus, from Figure 4.11 we can see that most of the instructions with DDI probability of 100% can be detected to be dead within very small instruction windows.

As expected, we observe that most of the dynamically dead instructions in our benchmarks are not always dead. We further analyzed the ratio of DDI that are dead with a very high probability of 90% and 70% and plotted these results in Figures 4.12 and 4.13 respectively for our optimized benchmark set. These results are much more optimistic. We find that, for our MiBench and SPEC benchmarks, 5.07% (65.76% of DDI) and 5.65% (55.83% of DDI) of total executed instructions can be detected to be dead, on average, with the probability of 90% respectively. With a probability of 70%, we can detect 5.51% (71.47% of DDI) and 5.99% (59.19% of DDI) of the dynamically executed instructions to be dead for our MiBench and SPEC benchmarks respectively. Interestingly, most of these are detected dead within small instruction windows, which can benefit some plausible hardware-based DDI detection techniques.

Figures 4.14, 4.15 and 4.16 plot similar graphs for DDI that are dead with probabilities of 100%, 90%, and 70% and within the illustrated instruction windows for the *unoptimized* benchmarks respectively. These results are mostly consistent with our earlier observations from the optimized benchmarks. On average over our benchmarks, while few dead instructions are always dead, a majority of DDI can be detected to be dead with high probabilities and relatively quickly within small instruction windows.

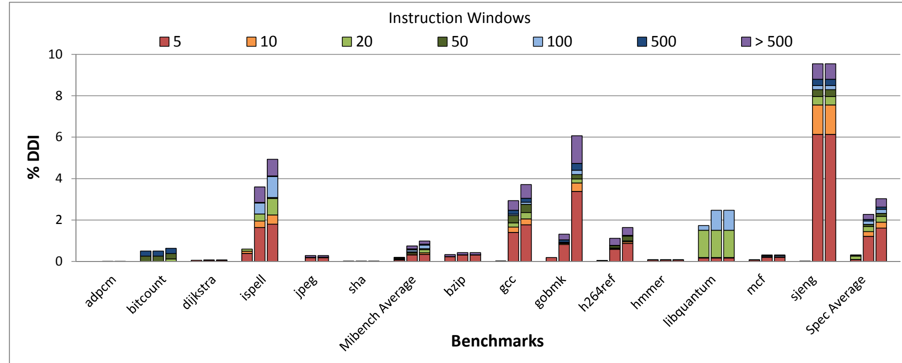


Figure 4.14. Percentage of DDI that are dead with 100% probability in the unoptimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information

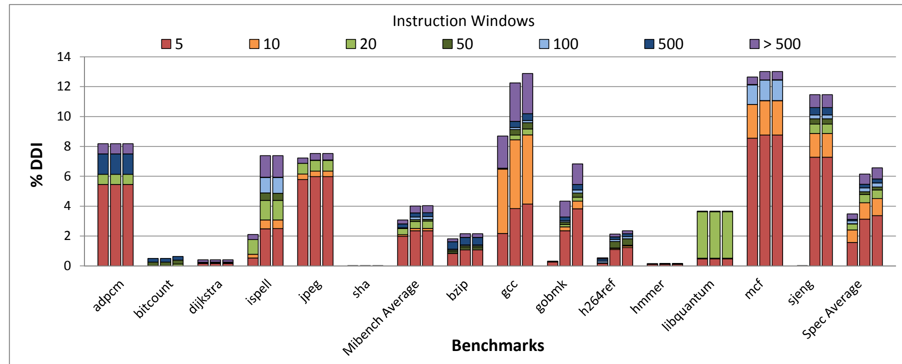


Figure 4.15. Percentage of DDI that are dead with 90% probability in the unoptimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information

4.1.2 Context Information to Improve DDI Detection

It is important to note that an instruction with a unique static (program counter or PC) location may be reached along different intra- and inter-procedural paths. Exploiting such compile-time and execution-time *context* information will effectively partition the dynamically executed instruction instances attributed to a single static location into multiple disjoint sets. While such partitioning may

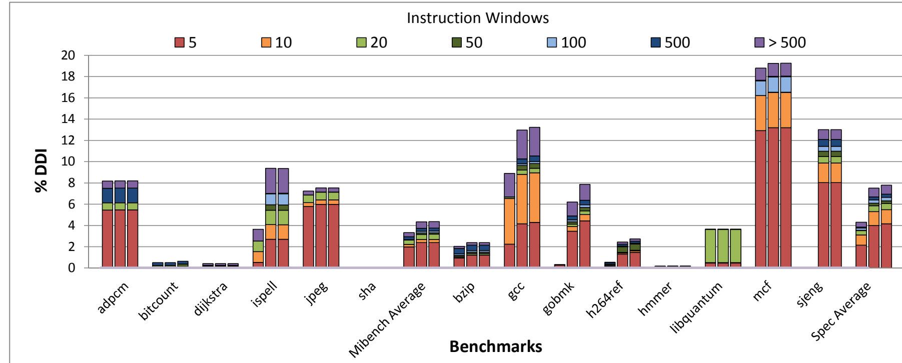


Figure 4.16. Percentage of DDI that are dead with 70% probability in the unoptimized x86 benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with context of a single (callee-static PC) function, and using the entire function stack for context information

increase the number of static locations a technique to eliminate DDI may need to track, it should also improve the probability of each instruction associated with context to be dead at run-time. In this section, we explore the impact of using such context information on the DDI probability and the number of instances that may need to be tracked.

There are many different kinds of context information that the compiler/hardware can exploit or track. For example, information contained in the dynamic function call-stack and/or the intra-procedural basic block path taken to an instruction can provide context to partition dynamic instruction instances. In this research, we limit the context information employed to the callee-static PC function call-stack.

To simulate and analyze the effect of gathering and using this context information, we employ our GCC compiler to add further instrumentation to the generated binaries. This new instrumentation dumps the static PC function identifier on the entry and exit of each function as it is reached during execution.

We also extended our trace algorithm implementation to maintain the necessary context information during the DDI analysis phase. We maintain knowledge

of the function context by employing a call-stack during our tracing algorithm. Remember that our algorithm scans the trace file backwards. Therefore, to generate the single callee-static PC function context information, we push the function name or identifier on the call-stack on encountering a function *return*, and later, pop the top element off the stack on reaching the function start. Thus, the single function context is easily obtained by only using the top callee-static PC function from the call-stack. To generate the full-stack context knowledge, we extend our call-stack implementation such that each stack location also holds a *hash* of the entire function call-stack state below it. On reaching the function return during our backward scan algorithm, we push a CRC32 hash checksum of the current function identifier with the previous checksum of top of the stack. The checksum value is restored on reaching the function start. Thus, the CRC32 hash allows us to cheaply maintain and employ context knowledge of the entire program call-stack state at each point.

As discussed in the last section, Figures 4.11, 4.12, and 4.13 show the percentage of DDI that are dead with 100%, 90%, and 70% probability in the optimized x86 benchmarks respectively. Remember that we obtain these probabilities by dividing the number of times the instruction instance is dynamically dead by the number of times the instruction instance is executed. Therefore, the ratio of DDI with probabilities of 90% and 70% (plotted in Figures 4.12 and 4.13 respectively) display the same or higher overall numbers than those plotted for a DDI probability of 100% Figure 4.11. The leftmost bar in each of these figures show the percentage of DDI without any context information. The middle bars display the percentage of DDI when using the context provided by a single immediate callee-static PC function. Finally, the rightmost bar for each benchmark presents the

percentage of DDI using the full callee-static PC function call-stack knowledge as context.

We can make several important observations by examining the graphs plotted in these three figures. First, we can see that the percentage of instructions falling into individual instruction windows can change (either increase or decrease) when context information is considered. Second, and much more importantly, we observe that context knowledge dramatically improves the fraction of DDI that are detected to be dead with a very high probability. Thus, using single callee-static PC function and full call-stack context, we find that, on average, for optimized SPEC benchmarks DDI that are dead with 100% probability increases from 1.73% to 3.25% (32.11% of DDI) and 4.38% (43.27% of DDI) respectively. Similarly, DDI with 90% probability rises from 5.65% to 7.61% (75.18% of DDI) and 8.13% (80.32% of DDI) respectively. On average, DDI for SPEC benchmarks with 70% probability increases from 5.99% to 8.66% (85.55% of DDI) and 8.92% (88.12% of DDI) respectively. It is very interesting and important to note that the simple single function context information is able to derive most of the benefits of using the full context information. Thus, using a small amount of context knowledge can significantly assist and benefit the task of various DDI detection/elimination algorithms.

Similarly, Figures 4.14, 4.15, and 4.16 also plot the impact of using context information on DDI probabilities of 100%, 90% and 70% respectively. Without discussing the specific numbers, we simply note that our observations from the earlier optimized benchmark results, namely that (a) a high ratio of static DDI instances are dead with a high probability, (b) this probability can be substantially improved by using little context knowledge, and (c) most DDI can be quickly

detected to be dead in small dynamic instruction windows, also hold very well over the unoptimized benchmark programs. Thus, although we only explore one avenue of context information, we can conclude that context knowledge can significantly improve the DDI probability. However, complex and large context knowledge is unnecessary because it is not likely to significantly increase DDI probabilities seen with a small amount of context information.

As noted earlier, context information improves DDI probability by partitioning the DDI instances that are attributed to a single static PC location into multiple PC-context locations. Therefore, it is obvious that using context knowledge can increase the number of locations that an online DDI detection or elimination algorithm may need to track, thereby increasing its cost and complexity. Figures 4.3, 4.4, 4.5, and 4.6 that were discussed earlier show the number and ratio of *static* instruction instances that correspond to the DDI as compared to the total number of static instructions reached during the execution of each x86 benchmark. The middle bar in these figures display these static instance numbers and ratios when using the single callee-static PC function context, while the last bar for each benchmark shows these when using the full callee-static PC function call-stack context knowledge.

We again find that the unoptimized and optimized x86 benchmarks show very similar trends. For the optimized SPEC benchmarks, and compared to the baseline of not using any context information, the number of static instances that attribute to DDI (and may need to be tracked) increases by about 2.56 times when using single function context (middle bars), and by 356.83 times when using the full function call-stack for context (last bars), on average. For the MiBench benchmarks, the corresponding increases are 1.40 times and 2.72 times respectively, on

average. Thus, as expected, using the full call-stack context information results in a massive jump, especially for the SPEC benchmarks, in both the number of *total* static instances as well as the number of instances corresponding to DDI. Much more important is the observation that, while employing even the single function context raises the number of static DDI instances, this increase is much more tempered and manageable. These results, combined with our earlier observation showing that using more complete context knowledge is not significantly beneficial, should bode well for the cost and complexity of future, simple online techniques to eliminate DDI.

4.1.3 Understanding and Characterizing Dynamically Dead Instructions

An important component of this project is to determine and understand the causes of DDI, so that effective techniques can be developed to eliminate them, when beneficial. Consequently, for this work, we manually analyzed the dead instructions for all our x86 MiBench benchmark programs to understand their deeper causes. Based on this analysis, we partition the DDI into seven distinct categories. These categories are selected such that dead instructions in each category could be addressed with one compiler or architecture-based solution.

We use the example code snippets in Figure 4.17 to explain some of the common instances of DDI that we encountered for these benchmarks. Our seven categories of DDI are described below:

1. **NOP instructions:** The NOP instruction does not change the state of the system. It is used for several purposes, such as to force memory alignment, to prevent hazards, etc. While NOPs are typically present in unoptimized

```
outp = (char *) outdata;
...
for(; len > 0 ; len-- ) {
   ...
   *outp++ = (delta & 0x0F);
}
...        // outp not used later
```

(a) increment of 'outp' in the last loop iteration is not used (adpcm)

(Category 4)

```
int temp =
for(k=1 ; k<DCTSIZE ; k++){
   if((temp = block[k]) == 0)
      ...      // path1: temp not used
   else
      = temp;   path2: temp is used
}
```

(b) 'temp' is partially dead along path 1 (jpeg)

(Category 5)

```
for(i=0 ; i<FUNCS ; i++){
   ...
   if(ct < cmin)
      cminix = i;
   if(ct > ctmax)
      cmaxix = i;
}
 = cminix;
 = cmaxix;
```

(c) Multiple sets of cminix and cmaxix in the loop not uses, except the last sets (bitcount)

(Category 7)

```
   for(p=w ; q=nword ; *p;)
      *q++ = mytoupper(*p++)
   *q = 0;
==> converted by compiler to
   //%edx=p; %ebx=w; %edi=q; %eax=0
   leal   -144(%ebp), %esi         // %esi=q[0] (nword[0]
   leal   -143(%ebp), %ecx         // %ecx=q[1] (nword[1])
   for(p=w ; *p){
      movzbl %dl, %edx              // %edx = (zero-extend) p
      movzbl hash+754(%edx), %edx   // %edx=mytoupper(*p)
      leal   (%ecx,%eax), %edi      // %edi = q++ (DDI)
      movb   %dl, (%esi,%eax)       // store %edx into mem. 'q'
      movzbl 1(%ebx,%eax), %edx
      add    $1, %eax               // access next array locs.
   }
   movb   $0, (%edi)               // use of %edi
```

(d) %edi set but not used in every loop iteration (ispell)

(Category 6)

```
   if(temp >= qval) temp/=qval;
   else temp = 0;
==> converted by compiler to
   xorl   %esi, %esi  // initialize temp to 0
   if(qval > temp){
    /* Compute temp /= qval into %eax */
      movl   %edx, %eax
      sarl   $31,  %edx
      idivl  -292(%ebp)

    /* Copy %eax to %esi */
      movl   %eax, %esi
   }
   movw   %si, (%edi,%ecx,2)
```

(e) Initialization of %esi wasted if the 'if–path' is taken (jpeg)

(Category 6)

Figure 4.17. Preliminary analysis of the occurrence of dynamic dead instructions

codes, we found that the compiler does a good job of eliminating all NOPs for optimized binary programs.

2. **Stack setup/adjustment:** The compiler grows the stack upon function entry in order to make space for local variables. However, in the typical code produced by GCC, both the base pointer and the stack pointer can be used to access local variables. We also found that the compiler often uses the base pointer to reference these locals and the stack register to only access the top of the stack. Therefore, if the top of the stack is never referenced in a function (like to push arguments onto the stack for a function call), then

the adjustment of the stack register upon entering the function becomes a dead instruction. Our trace algorithm will only see the stack register set consecutively, once to adjust the stack at the beginning of the function, and once to set the stack to the previous stack frame. The stack setup upon function entry is never used.

3. **Parameters not used in called function:** A common case of DDI are function parameters and return values that are never used. It may be possible in some cases for the compiler to determine this case, and use optimizations like *function cloning* to remove the dead instruction instances.

4. **Dead assignments in first/last loop iteration:** The example code from the *adpcm* benchmark in Figure 4.17(a) shows a common case of DDI, where a register or memory location is first used and then reset in each iteration of the loop (`outp`). The last set of such variables will be a dead instruction if it is not used after the loop ends. Optimizations such as loop peeling may be used to remove this DDI.

5. **Partial static dead instructions not removed by the compiler:** Figure 4.17(b) shows an example from the *jpeg* benchmark that illustrates the category of *partially dead instructions*. In this example, the initialization of variable `temp` outside the `for`-loop is dead. Furthermore, the set of `temp` in each loop iteration is also dead along path 1. It may be possible to update the *partial dead code elimination* optimization in GCC to resolve these cases of DDI.

6. **Introduced by compiler optimizations:** We also witness several examples of DDI introduced by compiler optimizations. Figure 4.17(d) shows

an example, where the register `%edi` is used to hold the current address of 'q'. The instruction '`leal (%ecx,%eax), %edi`' updates `%edi` in each loop iteration, but `%edi` is only used after the loop ends. Thus, all `%edi` updates, except the last, are dead. In another example, shown in Figure 4.17(e), the compiler attempts to reduce branch latency by moving the computation performed in the '`else`' portion of the `if`-branch before the branch, and then eliminates the '`else`' path to reduce the branch overhead. This extracted code will be dead at runtime if the `if`-path is taken. Additionally, the code example in figure 4.2 discussed earlier is another case of DDI that would fall into this category. Analyzing compiler optimization heuristics may be necessary to understand and eliminate this category of dead instructions.

7. **Deads that are difficult for the compiler to address:** Finally, Figure 4.17(c) shows code from the *bitcount* benchmark, where variables (`cminix` and `cmaxix`) are reset multiple times in a loop, but are only used after the loop ends. Thus, all except the last set of such variables are dynamically dead. Although easy to detect, it may be difficult for the compiler to automatically remove such DDI. We will explore microarchitectural techniques in future research, guided by compiler driven feedback, to remove such DDI.

Overall, we categorized at least 90% of the DDI in each benchmark. Instances of DDI that we did not categorize, the remaining 10% or less, will either fall into one of the seven categories described or, the dead instructions might not fall into any of them. The set of DDI we did not categorize is grouped into the *Miscellaneous* set. Figure 4.18 shows the contribution of each category of DDI for each benchmark. We can make several observations from this figure. First, all of our identified categories of DDI occur in multiple benchmarks. Second,

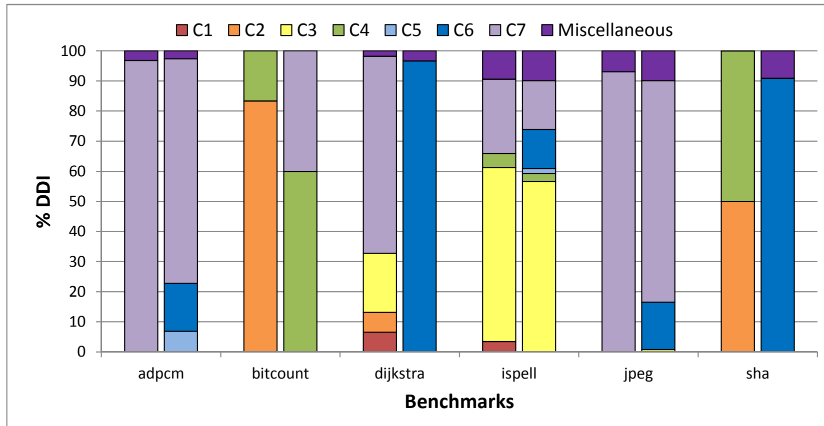


Figure 4.18. Relative categories of dead instructions in each benchmark. The left and right bars for each benchmark show DDI in unoptimized and optimized codes respectively.

compiler optimizations are able to completely remove *NOP* instructions from the generated codes. Third, the second category, dead instructions resulting from stack adjustments, does not exist in 90% of the optimized benchmarks. This does not mean that this type of DDI will not occur in optimized binaries, but it suggests the compiler makes an attempt at removing these useless instructions. One way the compiler might accomplish this is by utilizing registers as much as possible for local variables, instead of needing to push these variables onto the stack and requiring that initial stack setup. Fourth, the dead instructions for each benchmark fall into only a small number of categories, which may differ between its optimized and unoptimized versions. We may need to further refine these categories and/or add new ones as we explore other benchmarks and architectures in the future. Additionally, in future work, we will explore compiler and hardware techniques to study and resolve DDI for generated binaries and executed codes, as well as, their effect on the ratio of DDI and runtime performance of the program.

4.2 ARM Results and Analysis

While the x86 is still the prominent architecture for desktop and server-class machines, the ARM architecture is the de-facto standard for medium and high-end embedded/mobile devices. Therefore, any study for contemporary architectures cannot ignore the ARM architecture. In this section, we describe the results of our DDI analysis on benchmarks compiled for the ARM architecture.

We updated the ARM port of our GCC compiler to instrument binaries to collect DDI statistics on our ARM-Linux based systems. Our ARM PandaBoard machine has a dual-core ARMv7 Processor and is running the Ubuntu 10.10 sever OS. We use the same algorithm described in Section 3.2 to gather and traverse the execution traces for the ARM benchmark binaries and collect the number and characteristics of each program's dynamically dead instructions. Since the ARM is still characterized as an embedded architecture, we only use our MiBench benchmarks for our analysis in this section.

4.2.1 Implementation and Challenges

In this section, we first describe some implementation details and interesting challenges that we encountered in building and deploying our GCC-based framework to analyze DDI statistics on the ARM. While some challenges were expected due to the differences between the CISC x86 and RISC ARM architectures, other issues were more surprising and unanticipated. For example, the ARM architecture lacks powerful instructions like the '`lea`' instruction, or the load effective address instruction, that allows us to easily instrument the binary code to obtain the memory address used by load/store instructions on the x86. Instead, we had to more carefully study the memory addressing modes on the ARM and use a

greater number of more primitive instructions to get an accurate trace of memory addresses reached during execution on the ARM.

Predication, or conditional instruction execution, is an important, unique, and useful feature of the ARM architecture. However, the use of predication also has important ramifications on DDI analysis. As mentioned, an instruction is dynamically dead if it was executed by the processor and the result of that instruction is never actually used by the program. Therefore, to perform accurate DDI analysis, we need to know whether the predicated instructions encountered during execution are executed or not. A predicated instruction on the ARM executes conditionally based on the state of the CPSR (Current Program Status Register) register. If the condition is satisfied, the instruction is executed, and would be considered a DDI if its result is not used by the program. Otherwise, the instruction is effectively turned into a *NOP* instruction. We consider such instructions that fail their predicate condition to be a *predicated dead* instruction.

GCC RTL representation includes information on whether or not the assembly instruction is predicated. To accurately handle the issue of predicated instructions, we insert additional code instrumentation to dump the value of the CPSR register prior to executing such predicated instructions at run-time. The value of the CPSR register allows us to determine whether or not a predicated instruction was executed or converted into a *NOP*. Then, in our trace algorithm, we perform the same check of the CPSR register for every predicated instruction and determine its influence on the overall DDI value.

We also encountered several unanticipated challenges during our research on the ARM platform. For instance, the ARM is a lot slower and a somewhat more restrictive development platform compared to the x86. Furthermore, we only

had access to one ARM PandaBoard. Therefore, to reduce development time, we decided to perform as much of the development activity as possible on the x86, even for experiments on the ARM. While this turned out to be the right decision and significantly lowered our overall development time and cost, this approach required us to initially build a cross-compiler for the ARM on the x86. We discovered that building this cross-compiler toolchain is not a trivial task. The process is long, requires many different components, and correctly configuring these components is a complex activity.

Another issue we encountered, unlike our experience with the x86, was the discovery that the ARM port of GCC does not strictly restrict each RTL to have a one-to-one correspondence with the generated assembly instructions. Figure 4.19 displays two examples of RTLs created for the optimized ARM *ispell* benchmark, along with their corresponding assembly instructions. Figure 4.19(a) shows an

```
(insn 37 36 50 correct.c:678 (cond_exec (eq (reg:CC 24 cc)
      (const_int 0 [0x0]))
    (set (reg/v/f:SI 3 r3 [orig:190 q] [190])
      (reg/f:SI 13 sp))) (nil))
==> corresponding assembly instruction
  moveq  r3, sp
```

(a) RTL and corresponding assembly instruction of a conditionally executed instruction

```
(insn 174 173 181 correct.c:1269 (set (reg:SI 2 r2 [497])
    (eq:SI (reg:CC_NOOV 24 cc)
      (const_int 0 [0x0]))) (nil))
==> corresponding assembly instructions
  moveq  r2, #0
  movne  r2, #1
```

(b) RTL and corresponding assembly instructions where one instruction will always be executed and the other will be a predicated dead instruction

Figure 4.19. Example RTLs generated for the optimized ARM *ispell* benchmark with corresponding assembly instructions

example of an RTL that is typically generated to represent conditionally executed

instructions. Earlier, we discussed our method to handle predicated instructions and how we determine if the instruction actually executed. Our trace algorithm parses the RTL structure and checks for the `cond_exec` statement to determine if an instruction is a predicated instruction. Obviously, this test would return true for the example RTL in Figure 4.19(a).

However, Figure 4.19(b) is a special case. Firstly, it does not correspond one-to-one with a single assembly instruction. Secondly, the multiple corresponding assembly instructions are both predicated instructions. Thirdly, while the RTL does not contain the `cond_exec` statement, it still clearly uses the CPSR register, which corresponds to the RTL operand `(reg:CC_NOOV 24 cc)`, inside the `eq` operation. Finally, one of the corresponding assembly instructions will always be executed due to the fact that if the condition of one `mov` fails, then the other will be true. A machine dependent trace algorithm based on analyzing specific instruction sets might have examined the contents of the CPSR register twice in this example, once for each predicated `mov` instruction, to correctly handle this case. However, our decision to base our parsing algorithm on the GCC RTLs was to specifically avoid implementing a different algorithm for each architecture we choose to analyze. Since our instrumentation to print the contents of the CPSR register is based on the existence of the `cond_exec` statement in an RTL, our framework can not catch this instance. We assume the compiler creates a single RTL for cases like those in Figure 4.19(b) because only a single instruction will be executed regardless of the contents of the CPSR register. Therefore, the compiler does not need to include a `cond_exec` statement in the RTL.

However, we need a mechanism to accurately handle this special case in our trace algorithm. Fortunately, because our algorithm is not concerned with what

is inserted into the registers or memory locations (it is only concerned with what registers or memory locations are used and set), this type of RTL does not actually break our DDI detection scan. We still discover the correct number of DDI because we can determine from the RTL that `(reg:SI 2 r2 [497])` will be set and `(reg:CC_NOOV 24 cc)` will be used to set it. Unfortunately, for our purposes, we assume each RTL corresponds one-to-one with assembly instructions, which is how we determine a static instruction's contribution to the DDI found. Therefore, we still had to modify our trace algorithm in two ways. First, we need to keep track of the number of corresponding assembly instructions an RTL might have. Second, we have to sacrifice some accuracy when determining an instruction's contribution to the total DDI. For the example RTL in Figure 4.19(b), our trace algorithm would not be able to distinguish the predicated instruction from the executed instruction without checking the contents of the CPSR register value. Therefore, we choose to handle this scenario by always labeling the second assembly instruction as predicated dead. Again, while the occurrence of such RTLs does not change our ratio of DDI overall, it does make our analysis of a static instruction's contribution to the overall percentage of DDI less accurate. Fortunately, this is only a problem in the optimized ARM binaries and does not occur often.

We came across another challenge in the ARM port of GCC, which conflicted with the reason behind our decision to insert our instrumentation during the final code generation pass of the compiler. In theory, such late insertion of code instrumentation should have allowed optimizations to have been already applied to the code, which was what we found to be the case for the x86. Curiously, we found that our GCC's ARM port performs some ad hoc optimizations at a very

late stage of compilation during code generation. Such late optimizations interfere with our insertion of instrumentations, which cause some instrumentations to not always be inserted for every instruction that requires it. Again, fortunately, this was an issue that only occurred in the optimized binaries, and only very rarely. Currently, we eliminate this problem by manually inserting our instrumentation for these infrequent cases.

Finally, we encountered another unexpected challenge when we attempted to link some of our generated binaries. Because each instruction has a fixed width of 32 bits, the ARM is limited to loading unsigned immediate values of a certain size. Larger constants (such as memory addresses) require a data load from memory because they can not be stored in the instruction itself. Therefore, the ARM compiler embeds *literal pools* throughout the code to store these large constants. For each instruction that uses a large constant, the compiler replaces the constant with a short offset from the instruction location into the literal pool containing the constant. The compiler must ensure that these offsets are small enough to encode into each fixed-length instruction that requires them. As our instrumentation is inserted after the compiler has already placed the literal pools, it sometimes causes the code size to increase to a point where an instruction attempting to load a large constant into a register goes out of range of the literal pool. This causes the assembler to issue an error. In order to combat this issue, we placed additional literal pools throughout the code, ensuring that the offsets are small enough to encode into each instruction that performs a retrieval from the literal pool. We successfully overcame all these specific challenges on the ARM platform. Next, we present our DDI analysis and observations on the ARM. As mentioned earlier, we only use the embedded MiBench benchmarks for our results on the ARM.

4.2.2 Ratio of Dynamically Dead Instructions

Figure 4.20 shows the ratio of the number of total executed instructions for each benchmark that are dynamically dead. We can see that, on average, our ARM benchmarks contain a larger percentage of dead instructions than the previously discussed x86 binaries, including the SPEC benchmarks. Our optimized MiBench programs average 10.11% DDI, while the optimized benchmarks contain 20.60%. Again, this observation of optimized programs containing more dead instructions is consistent with earlier research performed on the Alpha architecture [2].

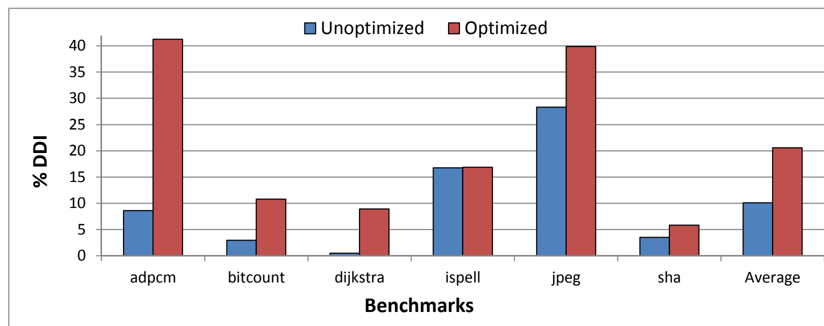


Figure 4.20. Percentage of dynamically dead instructions in ARM benchmark programs

While the x86 delivers a lower DDI ratio, the ARM benchmarks display a similarly high ratio of DDI to other architectures, such as the DEC Alpha [2, 4] and the Intel Itanium [17]. Again, this difference might be due to distinctions in the benchmarks, compiler, or the architecture selected for these works.

Figure 4.21 shows a similar chart seen in Figure 4.20, with the dynamically dead instructions broken up into four categories: dead instructions due to a register set, dead instructions due to a memory set, predicated dead instructions, and *NOP* instructions. As mentioned earlier, a predicated instruction is an instruction that executes conditionally based on the state of the CPSR (Current Program Status Register) register. If the result is true, the instruction is executed. Oth-

erwise, the instruction is effectively turned into a *NOP* instruction, which we consider a predicated dead instruction. On average the unoptimized MiBench benchmarks contain 8.44% register set dead instructions, 1.23% memory set dead instructions, 0.39% predicated dead instructions, and 0.05% *NOP* instructions. The optimized MiBench benchmarks contain 8.42% register set dead instructions, 4.63% memory set dead instructions, 7.55% predicated dead instructions, and 0% *NOP* instructions, on average.

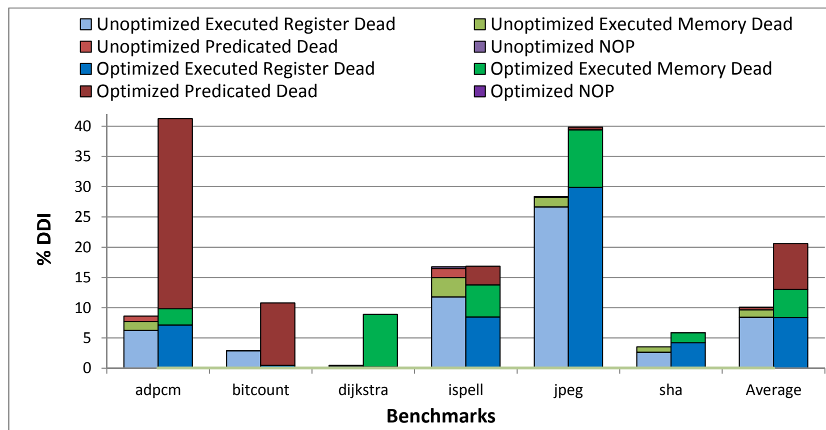


Figure 4.21. Percentage of dynamically dead instructions in ARM benchmark programs categorized as register set dead instructions, memory set dead instructions, predicated dead instructions, and *NOP* instructions

We can make several observations from this figure. First, all benchmarks contain both register and memory set DDI. Second, compiler optimizations are able to completely remove *NOP* instructions from the generated binaries. Third, all the benchmarks display a larger number of memory set dead instructions in the optimized binaries than in the unoptimized. And fourth, all the programs show the optimized binaries contain the same or more predicated dead instructions than the unoptimized binaries. The first three observations are patterns that we saw in our previous x86 results. However, the last observation is unique to the ARM

architecture because the x86 does not have predicated dead instructions.

4.2.3 Static Instructions and Dynamically Dead Instructions

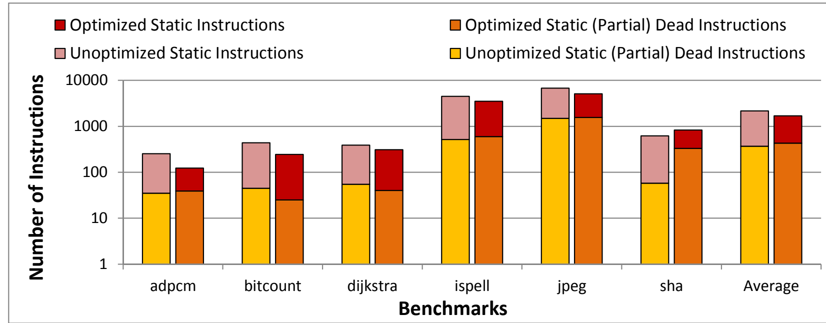


Figure 4.22. Number of instruction instances reached over execution of ARM MiBench benchmarks

Figure 4.22 displays the number and ratio of *static* instruction instances that correspond to the DDI as compared to the total number of static instructions reached during the execution of each ARM benchmark. Thus, we can see that on average, for the unoptimized MiBench benchmarks, about 13.34% of the static instruction instances contribute to the overall DDI. The average ratio grows for the optimized MiBench benchmarks, with about 23.50% of the static instructions contributing to the DDI. While we did not gather results with context information for these programs, this figure still allows us to make an important observation. On average, the unoptimized benchmarks contain more instruction instances reached over the execution, than the optimized benchmarks. This trend is consistent with the x86 results. We also observe that the percentage of static (partially) dead instructions significantly increases in the optimized ARM benchmarks. It is likely that this observation is due to the compiler's liberal use of predicated instructions in the optimized binaries.

Figures 4.23 and 4.24 are plotted to further study *only* the set of static in-

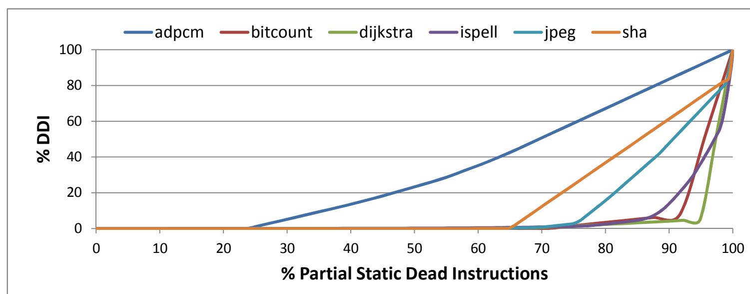


Figure 4.23. Contributions of static (partially) dead instruction instances to the DDI of optimized ARM MiBench benchmarks (sorted in ascending order)

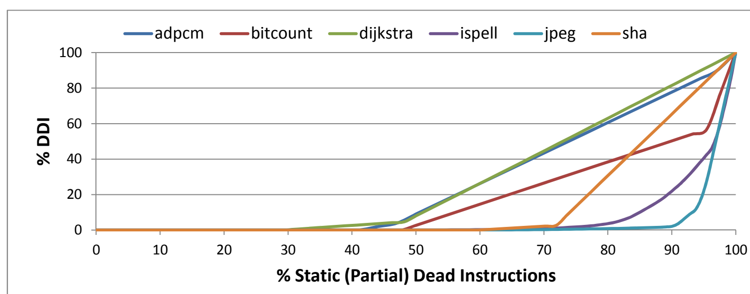


Figure 4.24. Contributions of static (partially) dead instruction instances to the DDI of unoptimized ARM MiBench benchmarks (sorted in ascending order)

structions that contribute to the program DDI. These figures sort (in ascending order) and display the contributions of individual static (partially) dead instruction instances to the overall percentage of dynamically dead instructions for the ARM benchmarks. Again, we can observe an important pattern in these figures: a small percentage of instruction instances actually contribute to the total DDI in our ARM benchmarks. The figures suggest that, for most of the benchmarks, whether they are unoptimized or optimized, about 70% of the static (partially) dead instructions do not contribute to 50% or more to the overall DDI. We observe a similar pattern in the x86 MiBench benchmarks.

Thus, in summary, the DDI characteristics and trends observed on the ARM are consistent with our earlier observations on the x86, with the important excep-

tion of the ARM binaries generating a significantly greater ratio of DDI. However, we also find that this increase in the DDI ratio over the x86 is primarily due to extensive use of the ARM-specific feature of predication by modern compilers like GCC.

Chapter 5

Future Work

Beyond the goals of this research to investigate the ratio, properties, and types of DDI, the larger objective of our project is to develop compiler and hardware techniques to resolve DDI, and evaluate their effect on program efficiency and power consumption for multiple contemporary compilers and architectures. Thus, there are several avenues for future work. First, we plan to study the effect of static techniques, including the use of different compilers and optimizations on DDI. Second, we will explore existing hardware-only [2] and develop new hardware and hybrid schemes to eliminate DDI on contemporary processors. Third, we will evaluate the potential of new device technologies, such as tunneling field effect transistors (TFET) [13, 15, 24] and spin-transfer torque RAM (STT-RAM) [6, 21, 22], that with their unique characteristics may enable innovative microarchitectural schemes to address the issue of DDI. Finally, the phenomenon of DDI is closely related to the issues of value locality, and ineffectual instructions that have also been widely studied by researchers. We plan to develop techniques to simultaneously deal with all these related problems in a uniform manner.

Chapter 6

Conclusions

As growth in single-thread program performance has stagnated in recent years, more aggressive software and hardware techniques are necessary to reverse this trend. Eliminating dynamically dead instructions that produce values not used by the program provides an approach that can not only improve single-thread program speed but also impact energy efficiency. Earlier studies report DDI to be significant, but conducted their experiments for now-defunct and less mainstream computer architectures. In this work we perform a comprehensive exploration of DDI properties that are important to DDI elimination techniques for the (currently) more relevant x86 and ARM processor architectures.

We first present our GCC-based instrumentation and analysis framework that provides a more portable environment to explore the number, ratio, and properties of dynamically dead instructions across different architectures. We discover that, for standard benchmark programs compiled with a state-of-the-art compiler, DDI comprises a significant fraction of the total executed instructions on both the x86 (SPEC – 10.12%, MiBench – 7.71%) and ARM (MiBench – 20.60%) processors.

As noted in earlier research, compiler optimizations typically have the effect

of increasing the DDI ratio. Further analysis shows that, while most of the static instructions corresponding to the observed DDI are only partially dead, a large fraction of such static locations are dead with a very high *probability* of over 90%. Importantly, we determined that online analysis can detect most DDI within small instruction windows. We also explored the percentage of static instructions that contribute to the overall DDI, and found that to be a relatively low number. Additionally, we investigated the effect of using *context* information to more precisely differentiate between the DDI instances attributed to a single static location, and found that a very limited amount of context knowledge can substantially improve DDI probability. Finally, we also present observations from our manual study to understand and categorize the DDI instances into a small number of independent sets. We believe that our results presented in this paper set the stage for much finer and deeper analysis, and eventual resolution of the problem of dynamically dead instructions for programs executing on contemporary machines.

References

[1] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the conference on Programming language design and implementation*, PLDI '98, pages 72–84, New York, NY, USA, 1998. ACM.

[2] J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 199–210, New York, NY, USA, 2002. ACM.

[3] S. CPU2006. Standard performance evaluation corporation (spec). http://www.spec.org/benchmarks.html, 2006.

[4] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 16–27, Washington, DC, USA, 2001. IEEE Computer Society.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[6] Y. Huai. Spin-transfer torque mram (stt-mram) challenges and prospects. *AAPPS Bulletin*, 18(6):33 –40, December 2008.

[7] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 147–158, New York, NY, USA, 1994. ACM.

[8] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM.

[9] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 182–191, New York, NY, USA, 2000. ACM.

[10] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 138–147, 1996.

[11] S. S. Lumetta and S. J. Patel. Characterization of essential dynamic instructions. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '03, pages 308–309, New York, NY, USA, 2003. ACM.

[12] D. Melski and T. W. Reps. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 47–62, London, UK, 1999. Springer-Verlag.

[13] N. Mojumder and K. Roy. Band-to-band tunneling ballistic nanowire fet: Circuit-compatible device modeling and design of ultra-low-power digital circuits and memories. *Electron Devices, IEEE Transactions on*, 56(10):2193 –2201, oct. 2009.

[14] C. Molina, A. González, and J. Tubella. Reducing memory traffic via redundant store instructions. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking*, HPCN Europe '99, pages 1246–1249, London, UK, 1999. Springer-Verlag.

[15] S. Mookerjea, D. Mohata, R. Krishnan, J. Singh, A. Vallett, A. Ali, T. Mayer, V. Narayanan, D. Schlom, A. Liu, and S. Datta. Experimental demonstration of 100nm channel length in0.53ga0.47as-based vertical inter-band tunnel field effect transistors (tfets) for ultra low-power logic and sram applications. In *Electron Devices Meeting (IEDM), 2009 IEEE International*, pages 1 –3, dec. 2009.

[16] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 153–167, New York, NY, USA, 1996. ACM.

[17] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 29–40, Washington, DC, USA, 2003. IEEE Computer Society.

[18] C. G. Nevill-Manning and I. H. Witten. Linear-time incremental hierarchy inference for compression. In *Proceedings of the Conference on Data Compression*, pages 3–11, Washington, DC, USA, 1997. IEEE Computer Society.

[19] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 138–148, Washington, DC, USA, 1997. IEEE Computer Society.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[21] C. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *High Performance Com-*

puter Architecture (HPCA), 2011 IEEE 17th International Symposium on, pages 50 –61, feb. 2011.

[22] Z. Sun, X. Bi, H. H. Li, W.-F. Wong, Z.-L. Ong, X. Zhu, and W. Wu. Multi retention level stt-ram cache designs with a dynamic refresh scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 329–338, New York, NY, USA, 2011. ACM.

[23] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *the international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 257–268, New York, NY, USA, 2000. ACM.

[24] K. Swaminathan, E. Kultursay, V. Saripalli, V. Narayanan, M. Kandemir, and S. Datta. Improving energy efficiency of multi-threaded applications using heterogeneous cmos-tfet multicores. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 247 –252, aug. 2011.

[25] S. Tallam, R. Gupta, and X. Zhang. Extended whole program paths. In *the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 17–26, Washington, DC, USA, 2005. IEEE Computer Society.

[26] A. Yoaz, R. Ronen, R. S. Chappell, and Y. Almog. Silence is golden? In *in Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[27] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 180–190, New York, NY, USA, 2001. ACM.

[28] Q. Zhao, J. E. Sim, W.-F. Wong, and L. Rudolph. DEP: detailed execution profile. In *15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 154–163, New York, NY, USA, 2006. ACM.