Understand and Categorize Dynamically Dead Instructions for Contemporary Architectures

Marianne J. Jantz and Prasad A. Kulkarni
Department of Electrical Engineering and Computer Science
University of Kansas, Lawrence, KS
Email: {mjjantz,prasadk}@ku.edu

Abstract

Instructions executed by the processor are dynamically dead *if the values they produce are not used by the program. Researchers have discovered that a surprisingly large fraction of executed instructions are dynamically dead. Dynamically dead instructions (DDI) can potentially slow-down program execution and waste power. Unfortunately, although the issue of DDI is well-known, there has not been any comprehensive study to understand and explain the occurrence of DDI, evaluate its performance impact, and resolve the problem for contemporary architectures.*

The goals of our research are to measure the ratio of DDI and systematically characterize them for existing state-of-the-art compilers and popular architectures, and then develop compiler and/or architectural techniques to avoid their execution at runtime. In this paper, we describe our GCC-based framework to instrument binary programs to generate control-flow and data-flow (registers and memory) traces at runtime. We present the distribution and percentage of DDI in our benchmark programs. We find that for the x86 platform, our embedded systems benchmarks compiled with GCC generally contain significantly fewer DDI than those observed in earlier research for other architectures. We also describe the outcome of our manual study to analyze and categorize the instances of dead instructions in our programs into seven distinct categories. We briefly describe our plan to develop compiler and architecture based techniques to eliminate each category of DDI in future programs. We believe that a close synergy between static code generation and program execution techniques may be the most effective strategy to eliminate DDI.

1. Introduction

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 [1, 2, 3, 4]. It has been seen that, on average, close to 14% and 20% of the instructions executed by programs on the Alpha and Itanium respectively are dynamically dead (even excluding NOP instructions) [4, 5]. It is obvious that executing dynamically dead instructions (DDI) will waste power and hardware resources, and likely slow-down the program execution. Consequently, earlier research efforts have explored both software and hardware approaches to address this problem. Traditional compiler optimizations, such as full and partial *dead code elimination* [6], were developed to *statically* eliminate all dead code [7]. 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 sophisticated compilers that contain and apply these optimizations. At the same time, attempts to address this issue with architectural and/or microarchitectural changes have not been adopted, likely due to high associated design and implementation costs [3, 1]. We believe that although the issue of DDI is well-known, old, and fundamental, it 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. Recently, the computing community is witnessing two major trends: the paradigm shift in hardware systems design [8] and the emergence of power as a first-class design constraint [9]. Today, physical barriers and technology limitations have significantly slowed the scaling of single-core program performance, which makes techniques to achieve automatic efficiency improvements for all existing and future microprocessors even more 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.

Consequently, novel strategies to effectively address the issue of DDI are essential for existing code generation and execution systems. Unfortunately, a major impediment to the development of new DDI elimination techniques is the lack of comprehensive knowledge regarding the characteristics of DDI in existing programs. Also problematic is the fact that even this sparse existing knowledge has only been gathered for (now) defunct (Alpha) or less mainstream (Itanium) computer architectures. Therefore, our goal for this project is to investigate the types and properties of DDI, and systematically characterize them for programs compiled using modern state-of-the-art compilers for contemporary and popular architectures. Additionally, we plan to use this knowledge to develop novel compiler and architectural approaches to effectively resolve each dominant category of dynamically dead instructions.

In this paper we describe the initial framework we built to detect, study, and categorize the DDI in benchmark programs. Our detection framework uses GCC to instrument the program with additional instructions to produce control-flow and data-flow traces on program execution. We have implemented algorithms to analyze the dynamic trace to determine the number and ratio of DDI, and their corresponding static instructions. We have manually studied the DDI in a few benchmark programs to better understand the causes for dead instructions. Based on this study, we propose static approaches to eliminate DDI from binary programs, and (micro) architectural techniques that can employ compiler-driven feedback to avoid the execution of each category of DDI at runtime. Thus, the major contributions of this work are the following:

- This is the first work to study the properties of DDI for contemporary architectures, like x86.
- We present the first categorization of DDI for optimized and unoptimized versions of programs compiled with modern compilers, such as GCC.
- This is the first measurement of the number and ratio of DDI for x86 benchmark programs.

The rest of this paper is organized as follows. We present background concepts and related work in the area of dynamically dead instruction detection and elimination in Section 2. We describe our GCC-based framework to detect and categorize DDI in Section 3. We present our experimental results in Section 4. Finally, we describe our future plans and the conclusions for this work in Sections 5 and 6 respectively.

2. Related Work

In this section we will describe background concepts and related work in the areas of characterizing dead instructions and compiler and architectural techniques for eliminating them. 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 [7]. 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 [6, 10].

Figure 1 presents examples to illustrate *fully dead* and *partially dead* code in programs. Figure 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 all program paths. The compiler removes such dead assignments from optimized codes. Figure 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 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 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 [6]. The second graph in Figure 1(b) illustrates the resulting code after applying this optimization. Thus, although modern compilers include sophisticated optimizations to eliminate all dead instructions, the prevalence of such instructions in typical benchmark programs suggests that they are not always successful. 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 [1]. 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

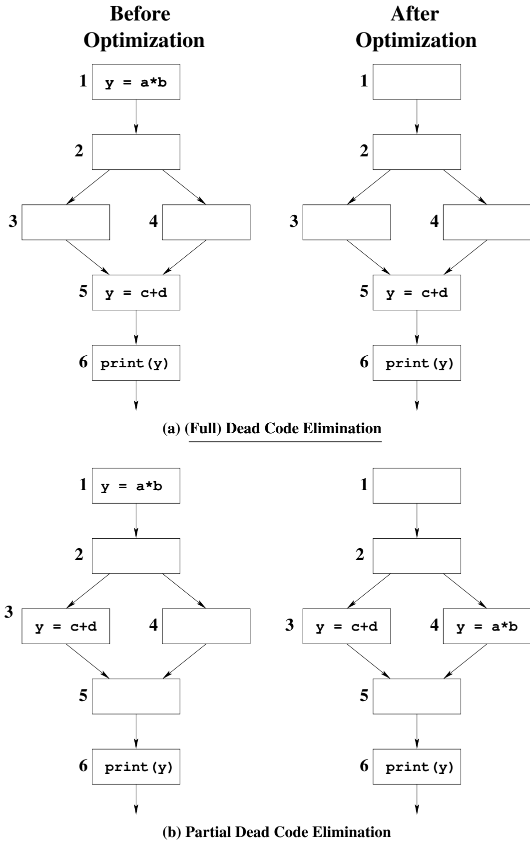


Figure 1. Varieties of dead code elimination optimizations in a compiler

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 [3]. 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.

Researchers have also explored static instructions that produce the same value on multiple consecutive dynamic invocations [11], or those that update a register or memory location with a value that it already contains [12]. 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 [13]. 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 [14, 15, 16]. The first algorithms for generating whole-program paths were presented by Larus [17] and Melski and Reps [18]. 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 [19, 20]. 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 [21, 22]. We will use and extend these algorithms to generate the control-flow and data-flow profiles for this research.

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. Our benchmark set consists of one program from each category of the MiBench benchmark suite [23]. The MiBench suite includes C benchmarks generally used in embedded applications. We employ and

modify the GCC compiler (version 4.5.2) for this research. The program is instrumented after all the optimizations are applied and immediately before code generation. The binaries are generated for 32-bit x86 platform. Each binary is natively executed to generate trace files, which we then analyze to discover important instances of DDI. In this section we provide further details on our compiler based implementations to generate and analyze execution traces to detect DDI in our benchmark programs.

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 plan 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 the optimization phases have been applied. We do not yet instrument library functions. Our tracing algorithm automatically marks all variables passed to a library function as being used, and we manually set the 'use' status of all array memory locations for the appropriate library calls. 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 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 [5, 1].

We use a simple example program in Figure 2 to illustrate the typical process of generating the dynamic program trace and analyzing it for dynamically dead instructions. Figure 2(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 2(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.

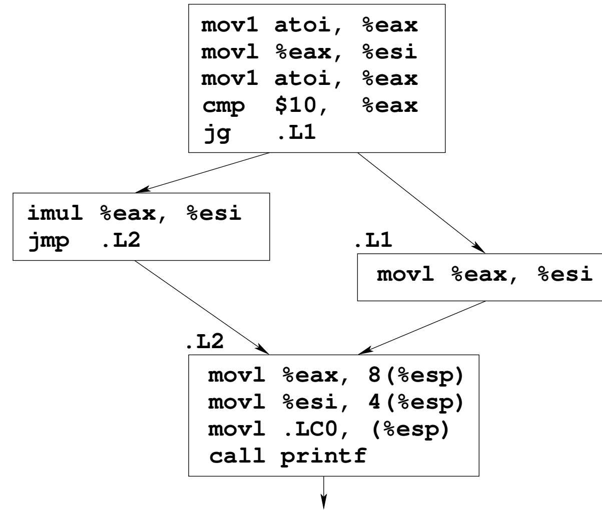
```
main(int argc, char *argv[])
{
    int i, j;

    i = atoi(argv[1]);
    j = atoi(argv[2]);

    if(j > 10)
        i = j;
    else
        i = i * j;

    printf("%d %d\n", i, j);
}
```

(a) Sample Input program



(b) Static program flow graph

`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 2. Sample example to illustrate the backward traversal algorithm to dead dynamically dead instructions

Figure 2(c) represents the dynamic trace that is generated on executing the binary program in Figure 2(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 2(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.

4. Experimental Results and Analysis

We use our modified version of GCC to instrument the benchmark programs to produce instruction and data traces at runtime, which we then analyze. For each benchmark, we generate two binaries, one that is unoptimized, and the other optimized with the GCC optimization flag set to *-O2*,

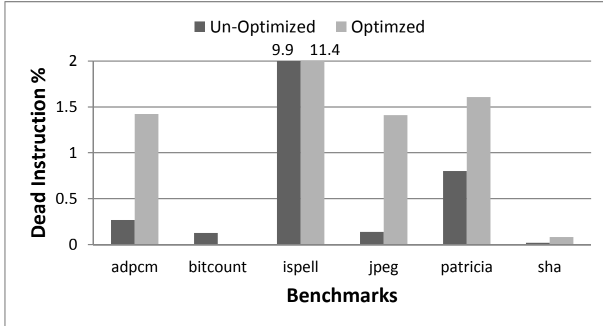


Figure 3. Percentage of dynamically dead instructions in benchmark programs

applying all available optimizations that do not involve a space-speed tradeoff. Thus, we produce two sets of results for each benchmark. In this section we present the results of our analysis regarding the ratio and characteristics of DDI for x86 binary programs generated by GCC.

4.1. Ratio of Dynamically Dead Instruction

We use the algorithm described in Section 3.2 to traverse the execution traces for each benchmark and collect the number of dynamically dead instructions. Figure 3 shows the ratio of the number of total executed instructions for each benchmark that are dynamically dead. We can see that most of our benchmarks only contain a small percentage of dead instructions. On average, our unoptimized benchmarks contain 1.9% of DDI, while the optimized benchmarks have a slightly higher fraction (2.67%). This observation of optimized programs containing more dead instructions is consistent with earlier research for different architectures [1]. However, our observations regarding the ratio of DDI are in stark contrast with earlier measurements on other architectures, such as the DEC Alpha [1, 5] and the Intel Itanium [4], that reveal a much higher ratio. This difference might be due to distinctions in the benchmarks, compiler, or the architecture selected for these works. In future work, we will investigate this issue by exploring the number of DDI with different and more benchmark, compiler, optimization, and architecture configurations.

4.2. 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. For this work, we manually analyzed the dead instructions for all our benchmark programs, and partitioned them into seven distinct categories. These categories were 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 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:** This 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.
2. **Introduced by compiler optimizations (not to reduce latency):** Figure 4(a) 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.
3. **Introduced by compiler optimizations (possibly to reduce latency):** In Figure 4(b), the compiler moves 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. Analyzing compiler optimization heuristics will be necessary to understand and eliminate these last two categories of dead instructions.
4. **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.
5. **Partial static dead instructions, not removed by the compiler:** Figure 4(c) 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. We will analyze and update the *dead code elimination* optimization in GCC to resolve these cases of DDI.
6. **Dead assignments in first/last loop iteration:** The example code from the *adpcm* benchmark in Figure 4(d) 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. Loop peeling may be used to remove this DDI.
7. **Deads that are difficult for the compiler to address:** Finally, Figure 4(e) 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, guided by compiler driven feedback, to remove such DDI [1].

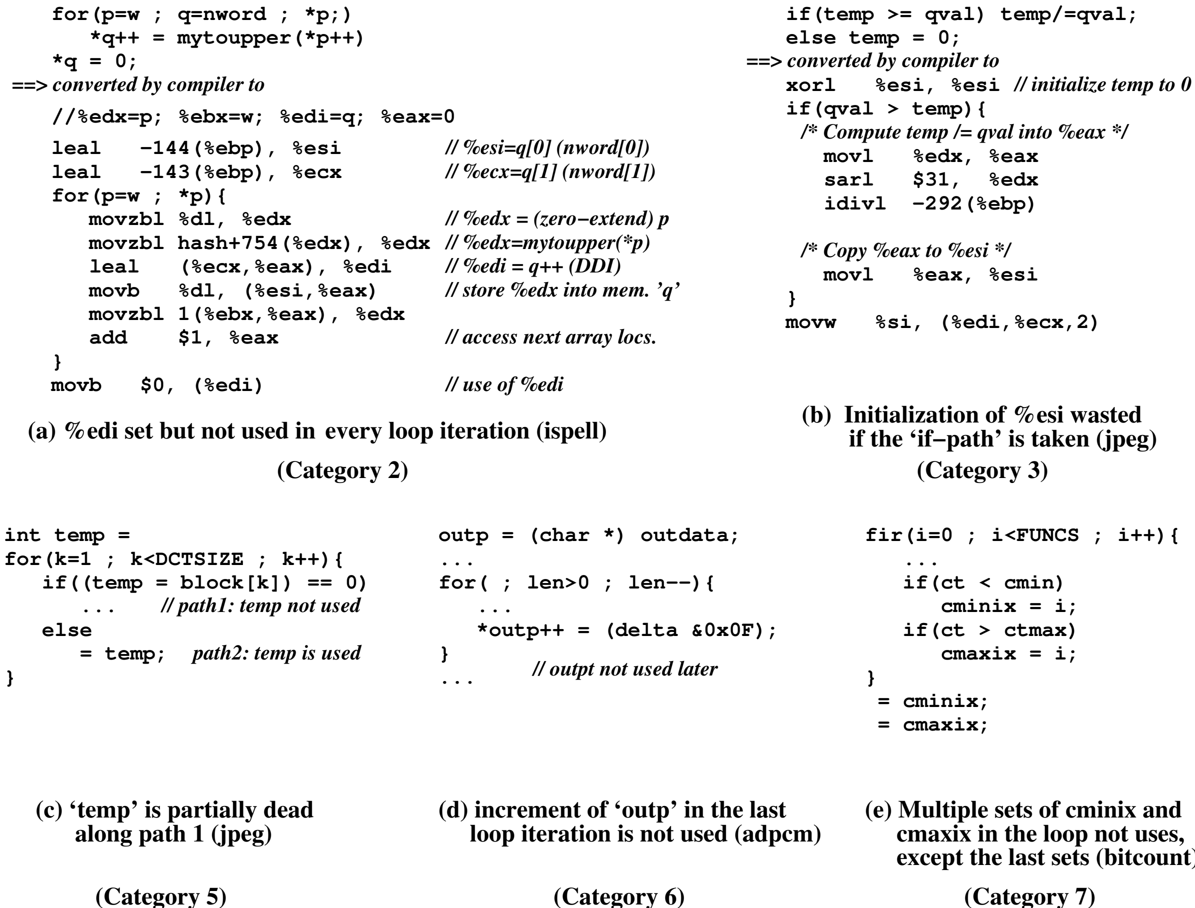


Figure 4. Preliminary analysis of the occurrence of dynamic dead instructions

Instances of DDI that do not fall into any of these categories are grouped into the *Miscellaneous* set. Figure 5 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, compiler optimizations are able to completely remove *NOP* instructions from the generated codes. Third, the dead instructions for each benchmark fall in 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 more and larger benchmarks 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.

5. Future Work

The larger goal of our project is to investigate the ratio of DDI, systematically categorize them, 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 will evaluate if our observations regarding the ratio and categories of DDI extend to more varied and larger benchmarks. Second, we plan to study the effect of multiple compilers, optimization configurations, and architectures on DDI. Third, we will extend our tracing framework with compression algorithms to allow smaller profiles, especially as we include larger benchmarks in our set. Finally, we also need to develop software and hardware algorithms to remove DDI statically or at runtime, and evaluate their effect on program performance.

6. Conclusions

In this work, we presented our GCC-based framework to determine the ratio of dynamically dead instructions in small embedded benchmark programs for the 32-bit x86-based systems. In contrast to previous results in the literature that find a large percentage of dynamically dead in-

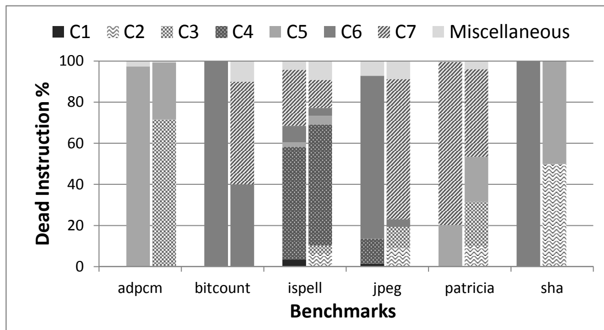


Figure 5. Relative categories of dead instructions in each benchmark. The left and right bars for each benchmark show DDI in unoptimized and optimized codes respectively.

structions for larger programs on Alpha (RISC) and Itanium (EPIC) processors, we discovered that for our set of benchmarks, DDI comprise only a small fraction of total executed instructions on x86 (CISC) systems. We note that this contrast may be the result of different architecture, compiler, and/or benchmark configurations, and we will further explore this distinction in our future work. Additionally, we also conducted one of the most detailed analysis of the detected dead instructions and showed that these can most often be categorized 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 modern machines.

References

[1] 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*, ser. ASPLOS-X, 2002, pp. 199–210.

[2] M. M. Martin, A. Roth, and C. N. Fischer, "Exploiting dead value information," in *Proceedings of the ACM/IEEE international symposium on Microarchitecture*, 1997, pp. 125–135.

[3] 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*, 2000, pp. 257–268.

[4] 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*, 2003, pp. 29–40.

[5] 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*, 2001, pp. 16–27.

[6] 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*, ser. PLDI '94, 1994, pp. 147–158.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[8] "International technology roadmap for semiconductors," accessed from http://www.itrs.net/Links/2008ITRS/Home2008.htm, 2008.

[9] T. Mudge, "Power: A first-class architectural design constraint," *Computer*, vol. 34, pp. 52–58, April 2001.

[10] R. Bodík and R. Gupta, "Partial dead code elimination using slicing transformations," in *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, ser. PLDI '97, 1997, pp. 159–170.

[11] 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*, ser. ASPLOS-VII, 1996, pp. 138–147.

[12] 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*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 182–191. [Online]. Available: http://doi.acm.org/10.1145/339647.339678

[13] 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*, 2003, pp. 308–309.

[14] 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*, 1996, pp. 153–167.

[15] G. Ammons and J. R. Larus, "Improving data-flow analysis with path profiles," in *Proceedings of the conference on Programming language design and implementation*, 1998, pp. 72–84.

[16] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997, pp. 138–148.

[17] J. R. Larus, "Whole program paths," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, 1999, pp. 259–269.

[18] D. Melski and T. W. Reps, "Interprocedural path profiling," in *Proceedings of the 8th International Conference on Compiler Construction*, 1999, pp. 47–62.

[19] S. Tallam, R. Gupta, and X. Zhang, "Extended whole program paths," in *the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005, pp. 17–26.

[20] 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*, 2006, pp. 154–163.

[21] C. G. Nevill-Manning and I. H. Witten, "Linear-time incremental hierarchy inference for compression," in *Proceedings of the Conference on Data Compression*, 1997, pp. 3–11.

[22] 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*, 2001, pp. 180–190.

[23] 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.