Properties of Dynamically Dead Instructions for Contemporary Architectures

Marianne J. Jantz, Katherine Wu and Prasad A. Kulkarni
Department of Electrical Engineering and Computer Science
University of Kansas
Lawrence, Kansas 66045
mariannejj@gmail.com, Katherine.Wu@ku.edu, prasadk@ku.edu

***Abstract*—Processor frequency scaling has greatly stagnated over the last few years, making it difficult to continue improving sequential or single-threaded program speed. Hardware and software system developers now need to devise innovative and aggressive schemes to grow sequential software performance. The goal of this work is to assess the potential and feasibility of eliminating *dynamically dead instructions* (DDI) – where the results of executed instructions are not used by the program – to benefit program speed. Specifically, we quantify the ratio of DDI in the dynamic instruction stream for different classes of contemporary programs (general-purpose vs. embedded) and architectures (CISC vs. RISC), and explore characteristics of DDI to assist the design of effective solution mechanisms.**

To achieve our goal, we develop a robust and portable compiler (GCC) based framework for DDI research, and target this investigation at contemporary x86 and ARM based machines. We find that while a substantial fraction of instructions executed by all classes of programs are dynamically dead, architectural features show a visible impact. Our experiments reveal that a handful of static program instructions contribute a majority of DDI. We further find that DDI are often highly predictable, can be detected within small instruction windows, and a small amount of static *context* information can significantly benefit DDI detection at run-time. Thus, our research can induce the development and adoption of practical DDI elimination techniques to scale sequential program performance in future processors.

Keywords*—*Dynamically dead instructions; architecture; compiler;

I. Introduction

An instruction executed by a processor is *dynamically dead* (DDI) if its calculated result is not used by the program [1]. It is obvious that executing dynamically dead instructions will waste power and hardware resources, and likely slow-down the program execution. Consequently, compiler optimizations, such as *dead and partial dead code elimination*, were specifically designed to remove such useless instructions from generated codes. In the past, researchers have observed that, even after applying such compiler transformations, a significant fraction (24% on the Alpha [2] and 20% on the Itanium [3], on average) of executed instructions are dead. However, existing compilers and hardware typically do not yet implement any dedicated techniques to detect and eliminate the DDI missed by the standard compiler optimizations.

Past DDI studies were performed in the era of exponentially growing uniprocessor clock speeds, when single-threaded applications were enjoying free, regular, and rapid performance gains, and processor real estate was more limited. More recently, physical barriers and technology limitations have effectively ended the rapid scaling of processor frequencies to automatically increase single-threaded software speed. We also find that many legacy programs and software tasks still employ sequential algorithms (such as finite state machines [4] and iterative numerical methods) that derive little benefit from increasing processor counts on newer multi-core machines. Additionally, even most parallel workloads are limited by their sequential components, as dictated by Amdahl's law [5]. Therefore, we believe that it will become attractive as well as important in the future to investigate and deploy more aggressive techniques to improve performance for sequential program components as well as for programs that cannot be recompiled/parallelized. Based on past research results, eliminating DDI is one such promising avenue.

Past DDI research has not conducted an extensive study of the characteristics of DDI. Earlier studies were performed on architectures (like the Alpha and the Itanium) that are either defunct or less mainstream today. Additionally, the impact of the instruction set/family on the fraction of DDI in executed programs has not been studied. Instruction set design decisions ranging from CISC vs. RISC to localized features, like predication, may affect the prevalence of DDI. Likewise, compiler optimizations can also influence the ratio and features of DDI in generated codes. Therefore, for any DDI elimination schemes to be deployed in existing systems, it is necessary to first study the extent and characteristics of the problem for contemporary architectures (such as the x86 and the ARM) and for programs generated using compilers that use the full suite of modern optimizations. Such a study will assist the development of practical solutions to the DDI issue, and reveal the potential performance benefit.

The goal of our research is to investigate the issue of DDI in the context of current compilers (GCC) and contemporary architectures (x86 and ARM) to determine the benefit and potential of DDI elimination to improve sequential program speed, and to help design practical resolution schemes. We make the following contributions in this work. (a) We develop a unique GCC-based compiler framework to *portably* detect, study, and categorize the DDI for multiple different architectures. (b) We determine the number and ratio of dynamically dead instructions and the corresponding static instructions that contribute to the overall DDI for general-purpose and embedded benchmark programs. (c) We design experiments to study DDI characteristics to assist the development of future

hardware, software, and hybrid DDI elimination schemes. Such measurements include determining the size of the dynamic instruction window to detect a dead instruction at run-time, and the effect of using static *context* information to isolate instructions with a high probability of being dynamically dead.

The rest of this document is organized as follows. We present the related work in Section II. We describe our GCC-based framework to detect, analyze, and categorize DDI in Section III. We present our experimental observations in Section IV. Finally, we describe future work and the conclusions in Sections V and VI, respectively.

II. Background and Related Work

Unreachable and dead code is introduced by software developers into high-level language programs or by the compiler when 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 [6], [7]. Although these optimizations are highly effective, high rates of DDI persist even for programs generated by compilers that apply these optimizations.

Past research has explored the DDI issue and suggested hardware mechanisms to find and eliminate DDI. Lumetta and Patel found that, on average, 15% of executed instructions in the SPEC2000 benchmarks on the Alpha processor are dead [8]. They also measured an additional 10% of the instructions to be *NOPs*. Fahs et al. proposed the *rePLay* architecture to provide dynamic optimization support at the hardware level [2]. Their dynamic optimization system built upon the Alpha simulator discovered 24% of DDI, on average, and could eliminate about 10% of them. Butts and Sohi analyzed some important properties of DDI and proposed a hardware technique to predict and eliminate DDI at runtime [1]. Their technique was able to detect 79% of useless instructions in their benchmarks and achieved up to 9.6% speedup benefits. This work only studied instructions that produce dead register values, and chose to ignore dead memory stores, NOPs and prefetches. A recent work studied the prominent causes of DDI for embedded programs on the x86 [9].

These existing works do not perform the same and as comprehensive an investigation into the characteristics of DDI as we do in this work. Additionally, they each explored a single hardware and benchmark set, and often used architectures that are currently not in common use. Our current research focuses on gaining a more thorough understanding of the potential and properties of DDI for multiple contemporary architectures and benchmark domains to assist the development and tuning of existing and new remedial mechanisms.

Researchers have also explored issues that are similar to DDI, and produce useless executed instructions. Several studies have investigated static instructions that produce the same value on multiple consecutive dynamic invocations [10], or dynamic instructions that update a register or memory location with a value that it already contains [11], [12]. This phenomenon is called value locality. 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 [11], [13]. Many of these works also propose and evaluate speculative mechanisms to remove or eliminate such useless instructions.

Martin et al. also worked on an issue related to DDI and presented hybrid schemes to statically mark the last use of register values that the hardware can later track to eliminate unnecessary save and restore instructions at procedure calls and context switches [14]. Sundaramoorthy et al. proposed a new processor microarchitecture to simultaneously run 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 [15]. These works also did not focus on analyzing and understanding the occurrence of DDI.

Detecting and understanding dynamically dead instructions requires 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 [16]–[18]. The first algorithms for generating whole-program paths were presented by Larus [19] and Melski and Reps [20], and later extended by several others [21], [22]. We use and extend these algorithms to generate the control-flow and data-flow profiles for this work.

III. Framework for Exploring DDI

In this section we briefly describe our experimental framework to generate program execution profiles to detect and investigate dynamically dead instructions. We use and modify a single GCC compiler (version 4.5.2) source code to build binaries for a 32-bit x86 platform, and cross-compile binaries for the ARM. Each compiled program is instrumented after all the optimizations are applied and immediately before code generation. We do not yet instrument library functions. Our tracing algorithm automatically marks all arguments passed to a library function as being used.

Each x86 binary is natively executed on Intel(R) Xeon(R) based machine. ARM binaries are cross-compiled on the x86 and run on an OMAP4 Panda board with a dual-core ARMv7 processor. The inserted instrumentations produce two trace files on program execution. These trace files contain an uncompressed sequential list of the basic block numbers along with a list of memory addresses as they are reached/accessed during execution. These trace files are later used in a single sequential scan to analyze and discover instances of DDI in our benchmark programs. The trace is scanned in a reverse order to reduce the complexity of classifying dead instructions. In particular, when processing a specific instruction in the trace, reverse scanning allows the liveness value of all consumers of the instruction's result to be already known [1], [2].

We use programs from the MiBench [23] and SPEC CPU2006 benchmark suites [24]. 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 SPEC CPU 2006 benchmarks contain larger CPU-intensive general-purpose applications. We include the eight 'C' integer benchmarks from the SPEC CPU 2006 set for our experiments. While our x86 experiments use our complete benchmark set, we use only the embedded MiBench benchmarks on the ARM platform.

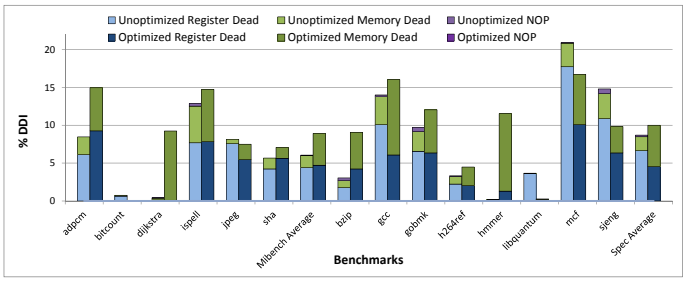

Fig. 1. Percentage of dynamically dead instructions in x86 benchmark programs. The DDI is further categorized as register set dead instructions, memory set dead instructions, and *NOP* instructions

```
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    // calc. index into array AdjMatrix
    movl  AdjMatrix(,%ecx,4), %esi   // Load AdjMatrix[iNode][i] into %esi
    cmpl  $9999, %esi    // Compare AdjMatrix[iNode][i] with NONE
    movl  %esi, iCost    // Mostly useless copy of AdjMatrix[iNode][i] into variable iCost
    je    .L45    // if false, enter 'if' statement
    ...
}
```

Fig. 2. Optimization to reduce load/store latency (dijkstra)

The *reference* data set for the SPEC benchmarks results in long (several hours) program run-times and large trace files. We employ the popular Simpoint mechanism to limit the program run-times with the SPEC benchmarks [25]. The Simpoint framework allows us to generate traces over smaller *representative* execution intervals instead of the entire program run. We used Simpoints to gather information for a maximum of five 100 million instruction windows for each SPEC benchmark. Our analysis considers all memory locations and registers as used upon starting the *backward* scan on each simpoint. Thus, our DDI numbers for the SPEC benchmarks reflect a conservative estimate.

IV. Experimental Results and Analysis

We use our modified 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, `-O0` and `-O3`, to generate the unoptimized and optimized binaries respectively. In this section we present the results of our analysis regarding the ratio and characteristics of DDI for x86 and ARM binary programs generated by GCC.

A. x86 Results and Analysis

We describe our DDI analysis results on the x86 platform in this section. Results of our experiments on the ARM are presented in Section IV-B. We use our backward tracing algorithm to traverse the execution traces for each benchmark (or benchmark's simpoints) and collect the number and characteristics of each program's dynamically dead instructions.

1) Ratio of Dynamically Dead Instructions: Figure 1 shows the ratio of the number of total executed instructions for each benchmark that are dynamically dead on an x86 machine. We see that most benchmarks contain a significant percentage of DDI, although not as much as was observed in earlier studies conducted on RISC (Alpha) and VLIW (Itanium) architectures. 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 slightly higher DDI. On average, 8.71% and 10.12% of the instructions executed by the unoptimized and optimized SPEC benchmarks respectively are dynamically dead on the x86. This observation of optimized programs containing more dead instructions is consistent with earlier research [1].

Figure 1 further breaks-up the DDI into three categories: dead instructions due to a register set, dead instructions due to a memory set, and *NOP* instructions. A *NOP* instruction is used for several purposes such as to force memory alignment and to prevent hazards, and does not change the state of the system. On average, the optimized MiBench benchmarks contain 3.88% register set dead instructions, 3.83% memory set dead instructions, and no *NOP* instructions. The corresponding numbers for unoptimized MiBench benchmark are 3.34%, 1.21%, and 0.06%, respectively. The optimized SPEC benchmarks contain 4.58% register dead instructions, 5.42% memory set dead instructions, and no *NOP* instructions. Finally, 6.64%, 1.85%, and 0.21% of instructions executed by the unoptimized SPEC benchmarks are dynamically dead due to redundant register sets, memory sets and *NOP* instructions respectively.

We note some interesting properties of DDI in our benchmark programs. First, all benchmarks contain both register and memory set DDI. Second, GCC compiler optimizations are able to completely remove *NOP* instructions from all x86 binaries. We also found that most benchmarks (with optimized bitcount being the only exception) display a larger number of memory set dead instructions in the optimized binaries than in their corresponding unoptimized variants.

Figure 2 shows an example of a common transformation applied by GCC that causes the program (*dijkstra*, in this case) to exhibit higher DDI after optimizations. This optimization assigns a local to a register in order to reduce the load/store latency. 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 instead. The `iCost` variable is still updated once in each loop iteration. Thus, while this optimization will likely reduce the load/store latency, the set (with no use) of the memory location is dead in all but (perhaps) the last loop iteration. This specific transformation actually accounts for 48.56% of the DDI found in the optimized *dijkstra* benchmark.

2) Static Instruction Contributions to DDI: Our analysis indicates that, in most cases, the *static* program instructions corresponding to the DDI are *partially* dead at run-time and/or difficult to remove using pure static techniques. While static schemes to eliminate DDI may be inadequate, they may still be able to assist run-time and hardware-based schemes to lower their overall hardware and power costs. For instance, we believe that a promising *hybrid* compiler-hardware approach may employ the compiler to tag instructions as *probably dead*,

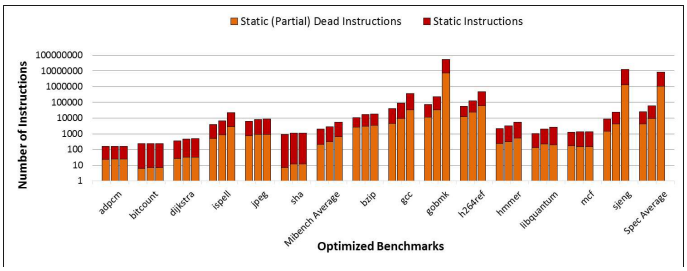

Fig. 3. Number of *static* instruction instances corresponding to DDI for **optimized x86** benchmarks. The three bars for each benchmark display static instructions reached without context information, with *single-call-site* context information, and *full-call-stack* context respectively. Note, the vertical axis is plotted on a logarithmic scale.

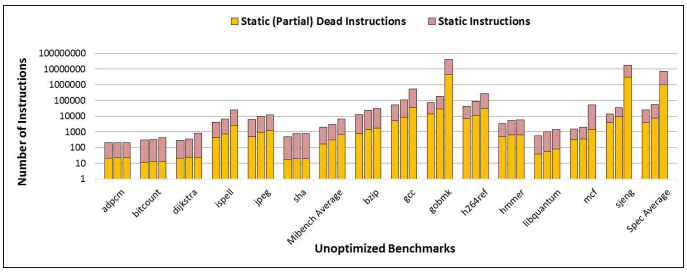

Fig. 4. Number of *static* instruction instances corresponding to DDI for **unoptimized x86** benchmarks. The three bars for each benchmark display static instructions reached without context information, with *single-call-site* context information, and *full-call-stack* context respectively. Note, the vertical axis is plotted on a logarithmic scale.

which will then be tracked by the hardware at run-time. For such techniques, success in eliminating DDI at low costs will depend on the compiler accuracy of tagging potentially dead instructions, the number of instructions that the hardware needs to track, and the instruction window that will be needed to determine the dead-*ness* status of an instruction at run-time. In this section we collect statistics to determine the feasibility and assist the development of such DDI elimination techniques.

First, we present results on the number of static instructions that generate DDI at run-time. The *first/leftmost* bar[1] for each benchmark in Figures 3 and 4 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 benchmark. Thus, we can see that, for optimized SPEC and MiBench benchmarks, only 18.62% and 8.97% of the static instruction instances, respectively and on average, contribute to the DDI. For our unoptimized benchmarks, 18.25% and 7.64% of the static instructions contribute to the DDI for SPEC and MiBench programs, respectively.

Figures 5 and 6 are plotted to further study *only* the set of static instructions that contribute to the program DDI.[2] These figures sort (in ascending order) and display the contributions of individual (partially dead) static instructions to the overall percentage of dynamically dead instructions for the optimized x86 benchmarks. We can observe an important pattern in these

[1]The remaining two bars for each benchmark will be discussed later.

[2]We only present plots for the optimized benchmark results to save space. The unoptimized benchmarks reveal similar trends.

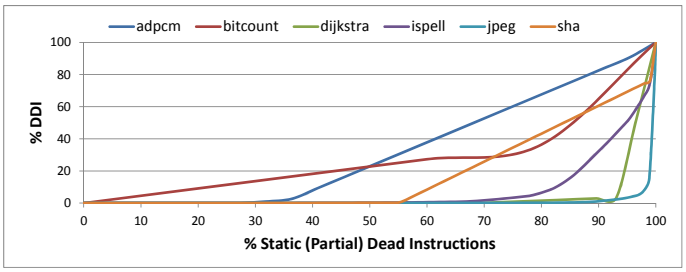

Fig. 5. Contributions of (partially dead) static instruction instances to the DDI of **optimized x86 MiBench** benchmarks (sorted in ascending order)

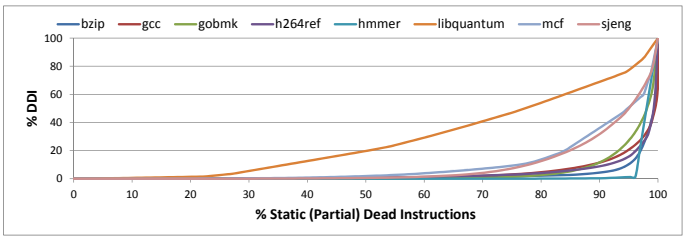

Fig. 6. Contributions of (partially dead) static dead instruction instances to the DDI of **optimized x86 SPEC** benchmarks (sorted in ascending order)

figures: a very small percentage of static instructions contribute a large majority to the total DDI in our x86 benchmarks. Thus, using compiler or profiling-based schemes to tag only these small number of *important* static instructions that contribute most to the DDI shows good promise to help run-time DDI elimination schemes.

Potential techniques to detect and eliminate DDI 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 program execution. Thus, if a specific static instruction is always dead at run-time, then we say that its DDI probability is 100%.

We further categorize these results based on how fast (in terms of number of intervening sequential instructions) an instruction is detected to be dead after it is reached. This detection *speed* may affect the length of the hardware instruction window maintained by the processor to detect dead instructions or how long a potential dead instruction will need to be delayed to avoid its execution for DDI elimination techniques. We extend 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. This modification enables us to find instructions detected dead within specific *instruction windows*. We employ (and plot) instruction windows of 5, 10, 20, 50, 100, 500, >500 to analyze the speed of detecting dynamically dead instructions.

We find that most of dynamically dead instructions in our benchmarks are not *always* (100% probability) dead. On average, for our MiBench and SPEC benchmarks, only 0.14% (2% of DDI) and 1.73% (17% of DDI) of total executed instructions are generated by static instructions that are always detected to be dead, respectively.

The *leftmost* bar for each benchmark in Figure 7 shows the percentage DDI that are dead with a high 90% probability

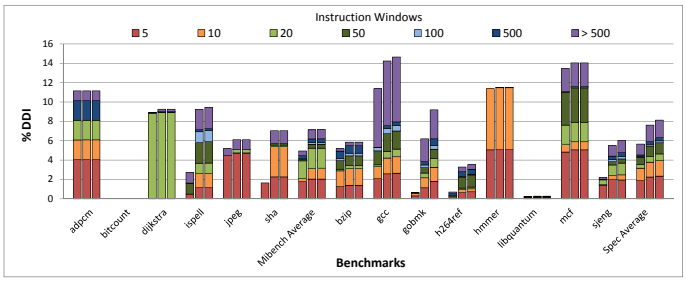

Fig. 7. Percentage of DDI that are dead with **90% probability** in the **optimized x86** benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with *single-call-site* context and *full-call-stack* context respectively.

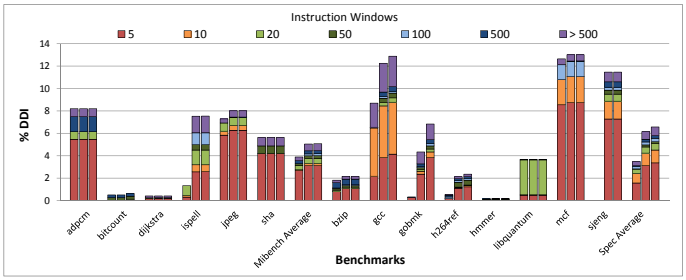

Fig. 8. Percentage of DDI that are dead with **90% probability** in the **unoptimized x86** benchmarks. From left to right, the bars for each benchmark display the percentage of DDI without context information, with *single-call-site* context and *full-call-stack* context respectively.

in the optimized x86 benchmarks. Thus, on average, 5.07% (65.76% of DDI) and 5.65% (55.83% of DDI) of dynamically executed instructions are generated by static instructions that are dead at least nine of every ten times they are reached for the MiBench and SPEC benchmarks, respectively. This figure also reveals that most of the instructions with DDI probability of 90% are detected to be dead within very small instruction windows, which can benefit some plausible hardware-based DDI detection techniques. Similarly, for 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. Again, most of these are detected dead within small instruction windows.

Figure 8 plots a similar graph for DDI that are dead with 90% probability and within the illustrated instruction windows for the *unoptimized* benchmarks respectively. These results (and those with 100% and 70% probabilities) are mostly consistent with our earlier observations for the optimized benchmarks. Thus, in summary, we can conclude that, while few dead instructions are always dead, a majority of DDI are detected to be dead with high probabilities and relatively quickly within small instruction windows at run-time.

3) Context Information to Improve DDI Detection: In this section we explore the potential and costs of a simple mechanism to improve DDI probability at run-time. This mechanism is based on the intuition that an instruction with a unique static (program counter or PC) location may be reached along different intra- and inter-procedural paths, which may display different DDI behavior. We exploit this *context* information to partition the dynamically executed instruction instances attributed to a single static location into multiple disjoint sets. While such partitioning may increase the number of static locations a technique to eliminate DDI may need to track, it should also improve the probability of each *instruction-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.

Approach: 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 (stack of) the PC-location of the function call that contains the dead instruction. To simulate and analyze the effect of using this context information, we modified our GCC compiler to add further instrumentation to the generated binaries. This new instrumentation identifies each function call by a distinct *call-site* identifier corresponding to the static (PC) location of the function *call* instruction. We appropriately extend our trace algorithm to maintain a stack of such context information during the DDI analysis phase. For our backwards tracing algorithm, the function call-site identifier is pushed on this call-stack on encountering a function return, and popped off the stack on reaching the function start.

For this work, we employ function call-site context information at two levels: (a) *single-call-site* – only the top stack entry is used, and (b) *full-call-stack* – the entire call-site stack is used. 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 call-site 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 on 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-site stack state at each point.

Benefit: The benefit of employing context information can be gauged from Figures 7 and 8. As discussed in the last section, Figure 7 shows the percentage of DDI that are dead with 90% probability in the optimized x86 benchmarks. Remember that we obtain these probabilities by dividing the number of times the static instruction instance is dynamically dead by the number of times the instruction instance is executed. 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 *single-call-site* context, while the rightmost bar for each benchmark presents the percentage of DDI using the *full-call-stack* knowledge as context.

Thus, from these figures, we observe that context knowledge can dramatically improve the fraction of DDI that are detected to be dead with a high probability. We find that using single-call-site and full-call-stack context, on average, for SPEC benchmarks DDI that are dead with 90% probability rises from 5.65% to 7.61% (75.18% of DDI) and 8.13% (80.32% of DDI), respectively. Similarly, for 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. 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. Importantly, simple single-call-site context knowledge is able to derive most of the benefits of using the full-call-stack context information.

Likewise, Figure 8 plots the impact of using context information on DDI with 90% probability for unoptimized benchmarks on the x86. We again find 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 for these experiments, we can conclude that context knowledge can significantly improve the DDI probability, and *simple context knowledge can get DDI probability benefits close to those achieved by more complex context collection algorithms*.

Costs: 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 3 and 4 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 benchmark. The middle bar in these figures displays these static instance numbers and ratios when using the *single-call-site* context, while the last bar for each benchmark shows these results when using the *full-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, we can see that the number of static instances that contribute to DDI (and may need to be tracked) increases by about 2.56 times when using single-call-site context (middle bars), and by 356.83 times when using the full-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 causes a large jump in both the number of *total* static instances as well as the number of instances corresponding to DDI, especially for the SPEC benchmarks. This jump will likely result in a corresponding increase in the cost of online DDI elimination algorithms. Fortunately, while employing even the single-call-site for context raises the number of static DDI instances, this increase in much more tempered and manageable. These cost results, combined with our earlier observation showing that using more complete context knowledge is not significantly more beneficial, should bode well for the cost and complexity of future, simple online or hybrid techniques to eliminate DDI.

B. ARM Results and Analysis

While the x86 is still the dominant architecture for desktop and server-class machines, the ARM architecture is fast becoming the de-facto standard for medium and high-end embedded/mobile devices. Moreover, the RISC-based ARM architecture presents a good contrast to the CISC x86, is more similar to the architectures used in earlier DDI studies (like the Alpha and MIPS), and may reveal the impact of architecture differences on observed DDI. 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, which we configured to run the Ubuntu 10.10 sever OS. Since the ARM is much slower as compared to the x86 processors, and characterized as an embedded architecture, we only use our MiBench benchmarks for our analysis in this section.

We appropriately extend the implementation of our GCC-based instrumentation framework to correctly handle the use of predication, or conditional instruction execution on the ARM. Predication is a distinctive feature of the ARM (and some other) architecture that is used to mitigate the costs typically associated with conditional branches. The use of predication has important ramifications on DDI analysis. To perform accurate DDI analysis, we need to know whether the predicated instructions encountered at run-time 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 each predicated instruction 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.

1) Ratio of Dynamically Dead Instructions: Figure 9 shows the ratio of the number of total executed instructions for each benchmark that are dynamically dead. We can see that our optimized and unoptimized MiBench programs contain 20.60% and 10.11% DDI, on average, respectively Figure 9 further breaks-up the dynamically dead instructions into four categories: dead instructions due to a register set, dead instructions due to a memory set, predicated dead instructions, and *NOP* instructions. 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 no *NOP* instructions, on average.

Thus, we find that, in contrast to programs on the x86 that we found deliver a low DDI ratio, the ARM bench-

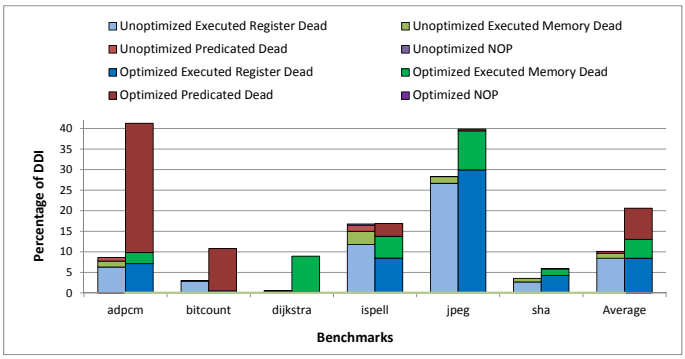

Fig. 9. Percentage of dynamically dead instructions in ARM benchmark programs categorized as register set dead instructions, memory set dead instructions, predicated dead instructions, and *NOP* instructions

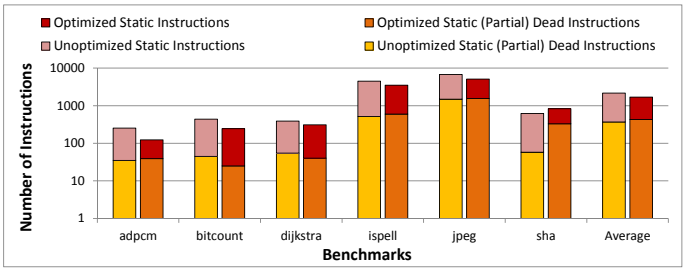

Fig. 10. Number of static instruction instances reached over execution of ARM MiBench benchmarks

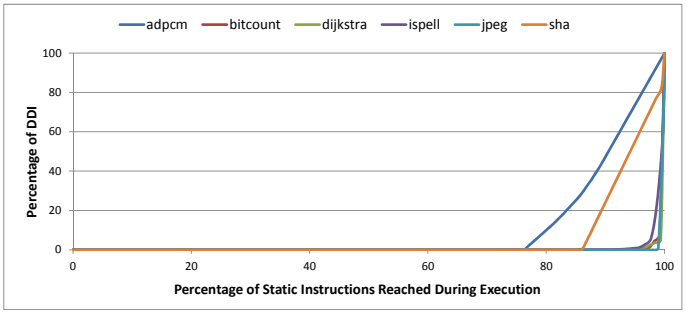

Fig. 11. Contributions of static (partially) dead instruction instances to the DDI of optimized ARM MiBench benchmarks (sorted in ascending order)

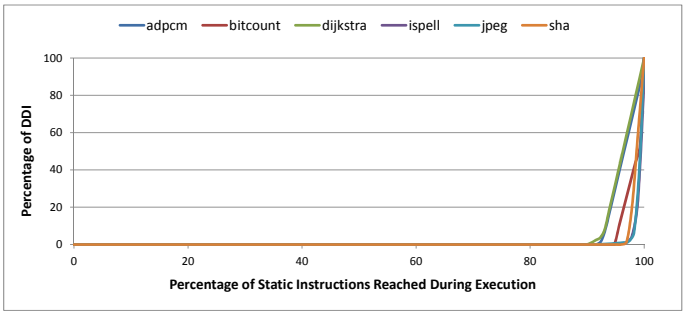

Fig. 12. Contributions of static (partially) dead instruction instances to the DDI of unoptimized ARM MiBench benchmarks (sorted in ascending order)

marks display a higher overall ratio of DDI similar to other architectures, such as the DEC Alpha [1], [2] and the Intel Itanium [3]. However, most of this increase can be attributed to the use of predication, which is a unique feature of the ARM that is absent in most other (studied) architectures. Many other observations we can make on the ARM are patterns that we also witnessed in our previous x86 results. For example, all benchmarks contain both register and memory set DDI, compiler optimizations are able to completely remove *NOP* instructions, and benchmarks display a larger number of memory set dead instructions in the optimized binaries than in their unoptimized variants. Likewise, optimized binaries on the ARM also typically contain more predicated dead instructions than the unoptimized binaries.

2) Static Instructions and Dynamically Dead Instructions: Figure 10 displays the number and ratio of *static* instruction instances that contribute to the DDI as compared to the total number of static instructions reached during the execution of each benchmark. Thus, we can see that on average, 13.34% and 23.50% of the static instruction instances contribute to the overall DDI for the unoptimized and optimized MiBench benchmarks respectively. We also find that, on average, there are more distinct instruction instances reached during the execution of unoptimized programs, as compared to the optimized benchmarks. This trend is consistent with the x86 results. We also observe that the percentage of static instructions contributing to DDI significantly increases in the optimized ARM benchmarks.

Figures 11 and 12 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 (partially dead) static instruction instances to the overall percentage of DDI for the ARM benchmarks. Again, we observe the pattern seen in the x86 benchmarks repeated in these figures: a small percentage of static instruction instances actually contribute most of the the total DDI in the ARM benchmarks. This observation may have important implications on techniques to benefit from DDI elimination.

In summary, DDI characteristics observed on the ARM are consistent with our earlier observations on the x86, with the important exception of the ARM binaries exhibiting 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.

V. Future Work

The larger objective of our project is to develop compiler and hardware techniques to eliminate DDI, and evaluate their effect on program efficiency and power consumption. 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 techniques [1] and develop new 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) and spin-transfer torque RAM (STT-RAM), 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.

VI. Conclusions

As growth in single-threaded program performance has stagnated in recent years, more aggressive 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 sequential program speed but also impact energy usage. However, before we invest in building techniques to eliminate DDI, we need a better understanding of the extent and properties of this issue for contemporary architectures, compilers, and benchmarks. The goal of our work was to perform this study.

We built our GCC-based instrumentation and analysis framework that provides a unique portable environment to explore the number, ratio, and properties of DDI across multiple target architectures. In contrast to earlier DDI studies conducted on different (RISC/VLIW) architectures, we discover that x86 displays a lower, though still significant, ratio of DDI to total executed instructions (SPEC – 10.12%, MiBench – 8.92%). On the ARM, DDI comprises a higher fraction of executed instructions (MiBench – 20.60%), but most of this increase over the x86 can be attributed to the use of predicated instructions. These DDI ratios set the upper-bound for the processor cycle count savings that can be achieved by eliminating DDI on these machines.

We perform experiments to understand DDI properties that can assist in the development of improved DDI elimination schemes. As noted in earlier studies, we also find that compiler optimizations often cause a small increase in the number of DDI. 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%. We determine that online analysis can detect most DDI within small instruction windows. We also find that a relatively low number of static instructions contribute to the overall DDI. We investigate the effect of using *context* information to better differentiate between the DDI instances attributed to a single static location, and find that a limited amount of context knowledge can substantially improve DDI probability. We believe that our results set the stage for much finer and deeper analysis, and eventual resolution of the problem of DDI for programs executing on contemporary 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*. ACM, 2002, pp. 199–210.

[2] 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*, ser. MICRO 34. IEEE, 2001, pp. 16–27.

[3] 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 *36th International Symposium on Microarchitecture*. IEEE, 2003, pp. 29–40.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: a view from Berkeley," EECS, University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, August 31 2006.

[7] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the 1994 conference on Programming language design and implementation*. ACM, 1994, pp. 147–158.

[8] 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*. ACM, 2003, pp. 308–309.

[9] M. J. Jantz and P. A. Kulkarni, "Understand and categorize dynamically dead instructions for contemporary architectures," in *Proceedings of the 2012 Workshop on Interaction between Compilers and Computer Architectures*. IEEE, 2012, pp. 25–32.

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

[11] K. M. Lepak and M. H. Lipasti, "On the value locality of store instructions," in *Proceedings of the 27th international symposium on Computer architecture*. ACM, 2000, pp. 182–191.

[12] 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*, ser. HPCN Europe '99. Springer-Verlag, 1999, pp. 1246–1249.

[13] 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.

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

[15] 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*. ACM, 2000, pp. 257–268.

[16] D. Mosberger and L. L. Peterson, "Making paths explicit in the scout operating system," in *Proceedings of the symposium on Operating systems design and implementation*. ACM, 1996, pp. 153–167.

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

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

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

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

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

[22] 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*. ACM, 2006, pp. 154–163.

[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.

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

[25] 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*. IEEE, 2001, pp. 3–14.