The Curious Case of Global Stable Loads

Shagnik Pal
The University of Texas at Austin
Austin, USA
shagnik@utexas.edu

Jeeho Ryoo
Fairleigh Dickinson University
Vancouver, Canada
j.ryoo@fdu.edu

Lizy K. John
The University of Texas at Austin
Austin, USA
ljohn@ece.utexas.edu

***Abstract*—A recent study presented an interesting statistic that about one-third and up to two-thirds of the load instructions in many x86 programs are redundant. These redundant loads fetch the same value from the same address throughout the execution of the program, and are called global stable loads (GSLs). We explore the behavior of GSLs across different architectures, compilers, and compiler optimizations. We also study GSLs in emerging ML workloads that are not explored in prior work. Our characterization of SPEC programs reveal a high locality in GSLs, that only 4.2-5.2% of static GSLs are responsible for 90-95% of the dynamic GSLs. We identify the reasons behind these dominant static GSLs, and posit that many of these GSLs are identifiable by the compiler prior to execution.**

In contrast to using a purely microarchitectural solution to eliminate GSLs, we consider a compiler-hardware co-designed solution wherein the compiler marks the likely stable loads, and the microarchitecture uses a small and fast cache to service these loads quickly and efficiently. We achieve a maximum speedup up to 7.43% for SPEC Float programs, and attain an energy reduction of 12.95% for the L1-D cache. We achieve these results using minimal hardware and without the need for a complex microarchitectural mechanism, yet the results are similar to state-of-the-art solutions.

***Index Terms*—Global Stable Loads, Compiler Techniques, Hardware Software co-design, Workload Characterization**

I. INTRODUCTION

Load instructions can cause data dependencies and cause pipeline stalls. A recent workload characterization by Bera and Ranganathan et al. [1] presented a startling statistic that on average 34.2% and up to 68.3% of load instructions in a program are redundant. These redundant loads are globally stable, i.e, they load the same value from the same address across the entire lifetime of the program. This high prevalence of such global stable loads is undesirable, especially in an era where optimizing instructions per cycle (IPC) is crucial, yet these redundant loads fetching a constant value unnecessarily occupy the pipeline, limiting performance gains from superscalar execution.

This discovery of **Global Stable Loads (GSLs)** raises a number of questions. Are they an artifact of x86 which has very few architectural registers, or are they present in high numbers in codes targeting other architectures as well? What about GSLs in code generated by compilers other than gcc? Prior work [1] illustrated a couple of scenarios where the compiler could not eliminate the GSLs, but it is unclear if they are the only causes responsible for GSL occurance. Furthermore, are GSLs universally occurring at high frequencies in emerging workload spaces, such as machine learning and AI?

Bera et al. [1] proposed a microarchitectural mechanism named *Constable*, that detects these GSLs dynamically, saves their data values, and skips their execution by reusing saved values. By eliminating address computation and redundant load data fetches, Constable improves performance by 5.1%, but at a cost of 12.4KB additional storage and necessitating a complex microarchitectural mechanism to identify and track these loads.

As well-understood, it is difficult for the compiler to register-allocate a frequently accessed global variable for the entire program scope pre-linking. Even if the compiler cannot eliminate GSLs, can it identify likely GSLs reliably enough to provide ISA hints that would enable the hardware to exploit/eliminate them in a simpler fashion than *pure* microarchitectural mechanisms? Hints (not mandatory instructions) can be inserted by the compiler even when it cannot determine with complete certainty that the particular variable under consideration is a constant across the entire program scope. A purely compiler mechanism has to guarantee functional correctness of the code, while there can be hint instructions to a microarchitecture, where if the hint was wrong, it affects performance but not correctness of the code. Is it possible to design a compiler-assisted hardware mechanism to yield nearly equal benefits?

We posit that in light of startling finding of GSLs, these questions require answers, and that the answers may guide future hardware or hardware-software collaborative mechanisms to address GSLs. In this paper, we quantitatively explore the questions posed above. Specifically, we make the following contributions:

- We study the prevalence of GSLs in codes produced by multiple compilers and targeting both the x86 and arm architectures, demonstrating that high GSL incidence is neither an artifact of x86 nor of one particular compiler.
- We examine the impact of various compiler optimizations [2] (e.g., function inlining) on GSL occurrences, and attempt to eliminate GSLs at compile-time via the application of specific optimizations. We conclude that there are no easy compile-time mitigations for GSLs.
- We study the disassemblies of the SPEC programs and identify the root causes of GSL occurance. Using these insights, we demonstrate how a compiler can provide hints to certain instructions being likely stable.

- We show that in the SPEC programs, only 4.2-5.2% of the globally stable load instructions are responsible for 90-95% of the programs' dynamic GSLs. This observation suggests that eliminating only a small percentage of the static GSLs in a program could achieve most of the performance benefit of a more comprehensive microarchitectural solution eliminating *all* GSLs.
- Leveraging above GSL locality, we evaluate the performance impact of a hardware-compiler collaborative approach using a small and fast cache dedicated to store the likely stable instructions identified by the compiler. Our solution achieves a speedup of 0.82% and 7.43% for SPEC Int and Float respectively, compared to the state-of-the-art (Constable) solution's reported speedup of nearly 1% and 5% respectively. In addition, we achieve 12.95% energy reduction in L1-D compared to Constable's 9.1%. All these results are obtained using a 6KB cache, half of Constable's hardware overhead of 12.4KB.
- Finally, we analyze the impact of GSLs across various machine learning (ML) workloads and observe that their dynamic occurrence is minimal, dropping below even 1% in large language models.

The remainder of the paper is organized as follows: Section II explains GSLs in greater detail. Section III describes the characterization methodology used in this paper. Section IV explores the incidence of GSLs across compilers and architectures, details the impact of various compiler strategies on GSLs, and quantifies the amount of locality in dynamic GSLs. In Section V, we identify some key factors contributing to occurrence of GSLs. Section VI proposes our hardware-software collaborative approach and section VII evaluates it. Section VIII examines the lower prevalence of GSLs in machine learning applications. Section IX summarizes previous work analyzing and mitigating instruction redundancy. Finally, section X concludes the paper.

II. Background

In this section we illustrate why GSLs occur. We consider *saxpy* ($y = a \cdot x + y$) for the simplicity of its load pattern: every y_i requires a read of a, x_i, and y_i. Figure 1a shows a C implementation of saxpy for a 10K-element vector, and figure 1b shows the corresponding disassembly produced at the default `-O0` optimization (in AT&T syntax with the destination last [3]). At the start of the saxpy function, all arguments (a, x, y and *array_size*) are loaded on the stack (green box). At every loop iteration (blue box), a, and base address of x are read once from the stack, and the base address of y is read twice (once to read y_i's value and once to write back the updated value). At the end of a loop iteration *array_size* is read from the stack and compared against current iteration number to check if looping should continue. Hence, these arguments constitute 5 loads at every loop iteration. Since the values of aforementioned arguments stay constant throughout the scope of the saxpy function, these loads are GSLs. The lines highlighted in red show the static GSLs. These 5 static GSLs over 10k loops

```c
void saxpy(float a, float x[],
    float y[], int array_size) {
    for (int i = 0; i < array_size; i++)
        y[i] = a * x[i] + y[i];
}
```

```c
void main() {
    int array_size = 10000;
    float a = 1.8; x[array_size], y[array_size];
    saxpy(a, x, y, array_size);
}
```

(a) saxpy: Computes y[i]=a*x[i]+y[i]

```
401745 <saxpy>:
  ...
                                                  Load arguments on to stack
40174d: movss   %xmm0,-0x14(%rbp)  ;Store %xmm0 (containing 'a') onto stack
401752: mov     %rdi,-0x20(%rbp)   ;Store %rdi (containing base address of 'x') onto stack
401756: mov     %rsi,-0x28(%rbp)   ;Store %rsi (containing base address of 'y') onto stack
40175a: mov     %edx,-0x18(%rbp)   ;Store %rdi (containing 'array_size') onto stack
40175d: movl    $0x0,-0x4(%rbp)    ;This stack location used for iterator 'i'
  ...
                                                  Loop iteration
401773: mov     -0x20(%rbp),%rax   ;Load base address of x. 10000 dynamic GSLs
401777: add     %rdx,%rax          ;%rax = x + i
40177a: movss   (%rax),%xmm0       ;%xmm0 has the value of x[i]
40177e: movaps  %xmm0,%xmm1        ;%xmm1 has the value of x[i]
401781: mulss   -0x14(%rbp),%xmm1  ;Load 'a', Multiply a*x[i]. 10000 dynamic GSLs
  ...
401793: mov     -0x28(%rbp),%rax   ;Load base address of y. 10000 dynamic GSLs
401797: add     %rdx,%rax          ;y[i] + a*x[i]  %rdx has a*x[i] and %rax has y[i]
  ...
4017ab: mov     -0x28(%rbp),%rax   ;Load base address of y. 10000 dynamic GSLs
  ...
4017ba: addl    $0x1,-0x4(%rbp)    ;Add 1 to iterator 'i'
4017be: mov     -0x4(%rbp),%eax    ;Load 'i' to %eax
4017c1: cmp     -0x18(%rbp),%eax   ;Load 'array_size', compare with 'i'. 10001 dynamic GSLs
```

(b) Unoptimized saxpy: The highlighted instructions show the static GSLs. They have a dynamic instance at every loop iteration

```
401745 <saxpy>:
  ...
  401754: movaps %xmm0,%xmm1          ;%xmm0 = a
  401757: mulss  (%rdi,%rax,4),%xmm1  ;%xmm1 = a*x[i]
  40175c: addss  (%rsi,%rax,4),%xmm1  ;%xmm1 =a*x[i]+y[i]
```

(c) Optimized saxpy: At higher compiler optimization levels, these GSLs are eliminated through register allocation.

Fig. 1: Understanding Global Stable Loads (GSLs) in disassemblies of saxpy under different optimizations.

result in 50k dynamic GSLs, i.e., 50k *redundant* instructions–32% of the total 157k dynamic loads in the program.

The `-O0` optimization level intentionally stores the variables on the stack so that the generated assembly code remains close to the source, and thereby simplifying debugging. However, higher optimization levels perform register allocation for the frequently accessed variables. Figure 1c shows disassembly for the same code at `-O1` optimization. In this case, the compiler has register allocated all the five constants and eliminated the redundant loads.

However, in more complex code, the compiler may be unable to eliminate similar GSLs due to register pressure. Register pressure occurs when the number of live variables exceed the available architectural registers. Register pressure in the surrounding scope could leave the compiler with insufficient architectural registers for variables it has identified to stay constant. This is particularly likely to happen when the variable has global program scope. Pointer arithmetic can also leave the compiler unable to reason about whether a variable is actually constant (and therefore register-allocatable). In either case, the compiler would not be able to eliminate the GSLs.

III. Methodology

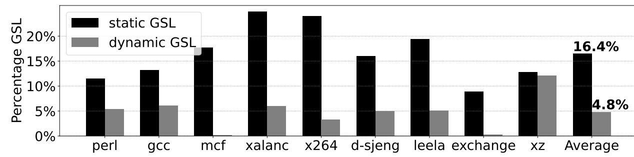
We characterized GSLs using Load-Inspector [4], a dynamic binary instrumentation tool based on Intel's Pintool [5]. We used the SPEC (Standard Performance Evaluation Corporation) CPU 2017 [6] Integer and Floating Point programs as our workloads. To analyze GSLs across compilers, we use gcc [7] version 11.4 and clang [8] version 14.0 as our compilers.

We also analyze GSLs across x86 and arm architectures. We use AMD EPYC 7742 [9] for x86. For arm, we ported the Load-Inspector code into gem5 [10], [11] simulator, and using gem5 as a functional simulator we simulate the SPEC binaries in syscall emulation mode with an atomic core and memory model. Unless stated otherwise, all the SPEC programs are compiled using gcc `-O3` optimization. We used the ChampSim [12] simulator for our microarchitectural explorations.

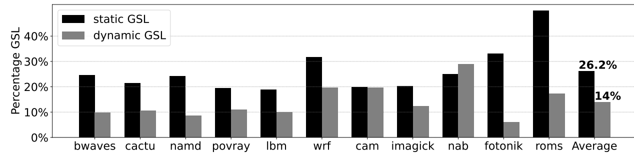
We also study GSLs in ML inference workloads. The challenge was that most of the ML models are written in python while Intel's pin-tool requires pre-compiled binaries. Our study comprised a combination of C++ based inference models such as Meta's llama2 [13], [14], OpenAI's whisper [15]–[17], darknet [18], and YOLO [19], [20]. We also characterize some ML models [21] written in python that were converted to binaries using nuitka [22], a python compiler.

IV. GSL Analysis

We define percentage static/dynamic GSLs as the number of static/dynamic GSLs divided by all static/dynamic loads. Figure 2 shows the GSLs in SPEC Int and Float programs. We see that the percentage of dynamic GSLs is considerably less than static GSLs, suggesting that not all static GSLs have high dynamic instances. In this section we dive deeper into these static and dynamic GSLs and see how they behave across different compilers and architectures.



(a) SPECrate Int: Average 16.4% static and 4.8% dynamic GSLs



(b) SPECrate Float: Average 26.2% static and 14% dynamic GSLs

Fig. 2: Static and Dynamic GSLs in SPEC programs.

A. *GSL across Compilers and Architectures*

TABLE I: Comparing GSLs for gcc and clang compilers

Workload	Parameter that's averaged over the programs	gcc	clang
SPEC Int	Static GSLs	16.45%	19.15%
	Dynamic GSLs	4.80%	4.54%
	Dynamic Loads	392B	377B
	Dynamic Instructions	1.2T	1.4T
SPEC Float	Static GSLs	26.52%	28.11%
	Dynamic GSLs	13.09%	14.99%
	Dynamic Loads	773B	718B
	Dynamic Instructions	2.2T	2.2T

Prior study [1] of GSLs has been done on x86 systems with a gcc compiler. In this section, we examine whether GSL behavior is an artifact of x86 or gcc, or whether it is universal across compilers and architectures.

GSLs across compilers: We run the SPEC Int and Float programs compiled on gcc and clang compilers. Figure 3 shows that there are a few programs which show reduction in dynamic GSLs on using clang (namd infact also reduces the dynamic loads). Table I shows the percentages of different microarchitectural parameters averaged across all the programs in the given workload. In SPEC Int, while the average static GSLs show slight variation with 16.45% and 19.15% for gcc and clang respectively, the percentage dynamic GSLs remain largely similar. SPEC Float as well shows very slight difference with clang exhibiting nearly 2% more static and dynamic GSLs percentages. Furthermore, the number of instructions and loads are also similar.

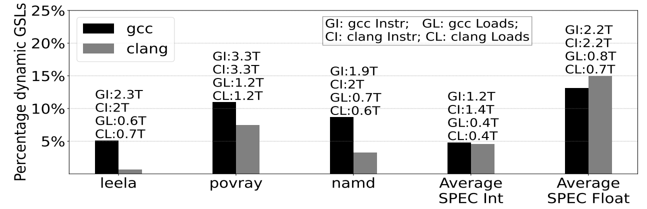


Fig. 3: SPEC programs with notable changes in dynamic GSLs from gcc to clang.

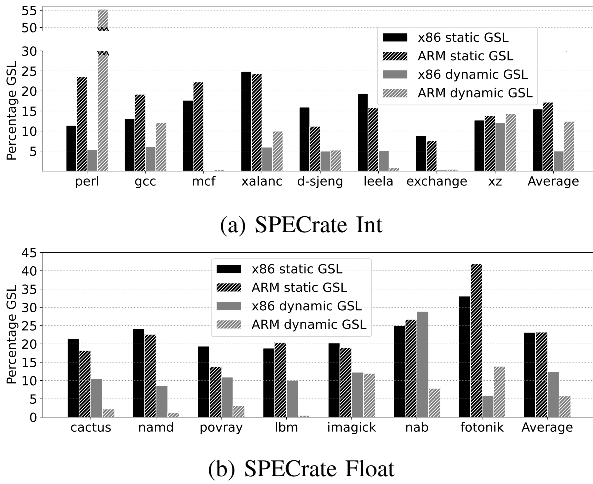
The gcc and clang compilers mainly differ in their optimization phases [23], but the Intermediate Representation (IR) and code generation phases are largely unchanged [24]–[26]. Therefore, the final assembly generated remains largely unchanged, resulting in no significant alteration of the overall stable behavior.

GSLs across architectures: Figure 4 shows the percentage of static and dynamic GSLs across arm and x86 architectures for SPEC programs. We see that the average static GSLs across SPEC Int and Float programs show very little variation. The average dynamic GSLs in arm are double in SPEC Int and half in SPEC Float, as compared to x86. Considering the SPEC workload as a whole, the percentage dynamic GSLs on arm and x86 remain similar at 8.49% and 9.27% respectively. Despite having more registers, arm does not show significant reduction in GSLs. Therefore, we infer that global stable behavior is not a characteristic of a specific architecture and exists across both x86 and arm.

B. *Limitations of Compiler Optimizations*

In this section, we attempt to eliminate GSLs using compiler optimizations. Our baseline is compiled with the most aggressive optimizations (`-O3`). We consider GSL causing scenarios, and use compiler flags to eliminate them.

Function Inlining: Bera et al. [1] had presented an example of function inlining causing GSLs: a function argument that is constant across repeated calls to the function, whose



(a) SPECrate Int

(b) SPECrate Float

Fig. 4: Comparing the GSLs in SPEC on arm and x86

TABLE II: GSLs across different Compiler Optimzations.

Workload	Optimization	Average Dynamic GSLs	Average Static GSLs	Average Dynamic Instr	Average Dynamic Loads
SPEC Int	(Baseline) Aggressive Optimizations (-O3)	4.80%	16.45%	1.17T	392B
	Disable Inlining (-fno-inline)	4.23%	16.75%	2.15T	642B
	Basic Optimizations (-O1)	4.78%	16.39%	1.46T	467B
	No Optimizations (-O0)	3.42%	14.83%	5.37T	1.71T
	Reduce Reg. Pressure	4.71%	16.39%	1.17T	391B
SPEC Float	(Baseline) Aggressive Optimizations (-O3)	13.09%	26.52%	2.17T	773B
	Disable Inlining (-fno-inline)	13.66%	26.26%	2.30T	792B
	Basic Optimizations (-O1)	16.38%	26.55%	2.45T	870B
	No Optimizations (-O0)	21.53%	28.11%	9.2T	3.77T
	Reduce Reg. Pressure	14.30%	26.11%	2.11T	731B

inlined invocation causes it to inherit the register pressure of its surrounding scope. Would it have been better for the compiler to avoid inlining the function, to possibly reduce register pressure, and avoid redundant loads? We use the gcc flag "-fno-inline" on top of our baseline to avoid the automatic inlining by the compiler. Table II shows that the static or dynamic GSLs do not change much but the number of dynamic instructions have increased (from 1.17T to 2.15T in SPEC Int) due to function call overheads. Thus, disabling function inlining is not enough to reduce the register pressure.

Changing Optimization Level: Similarly to function inlining, several other aggressive optimizations may increase GSLs. For example, software pipelining and loop fusion increase the number of live registers at a given time leading to higher register pressure. We vary the optimization levels from our baseline of most aggressive optimizations (-O3) to basic optimizations (-O1) to no optimizations (-O0). Table II shows that decreasing the optimization levels reduces the dynamic GSLs in SPEC Int (from 4.8% to 3.42%) and increases it in SPEC Float (from 13.09% to 21.53%). Additionally, the number of instructions and loads is significantly increased. Thus, simply using unoptimized code is not a solution.

Reduce Register Pressure: So far, we have observed that the primary cause of GSL occurrences is high register pressure. We now try to reduce register pressure using compiler flags. There are three gcc flags [2] that can be used to reduce register pressure on top of -O3 optimization: *-flive-range-shrinkage*, *-fsched-pressure*, and *-fira-loop-pressure*. Table II shows that the number of GSLs do not change with this optimization. Even the number of instructions or loads did not change much. The reason is, we cannot force the compiler to reduce the register pressure. GCC takes these directives as suggestions, but if the number of live registers far exceed the available architectural registers, the compiler cannot do anything.

Therefore, we conclude that we cannot eliminate GSLs just by changing the compiler optimizations. Even if we instruct the compiler force register allocation, it is limited by the available registers in the underlying architecture, so reducing register pressure may not be possible.

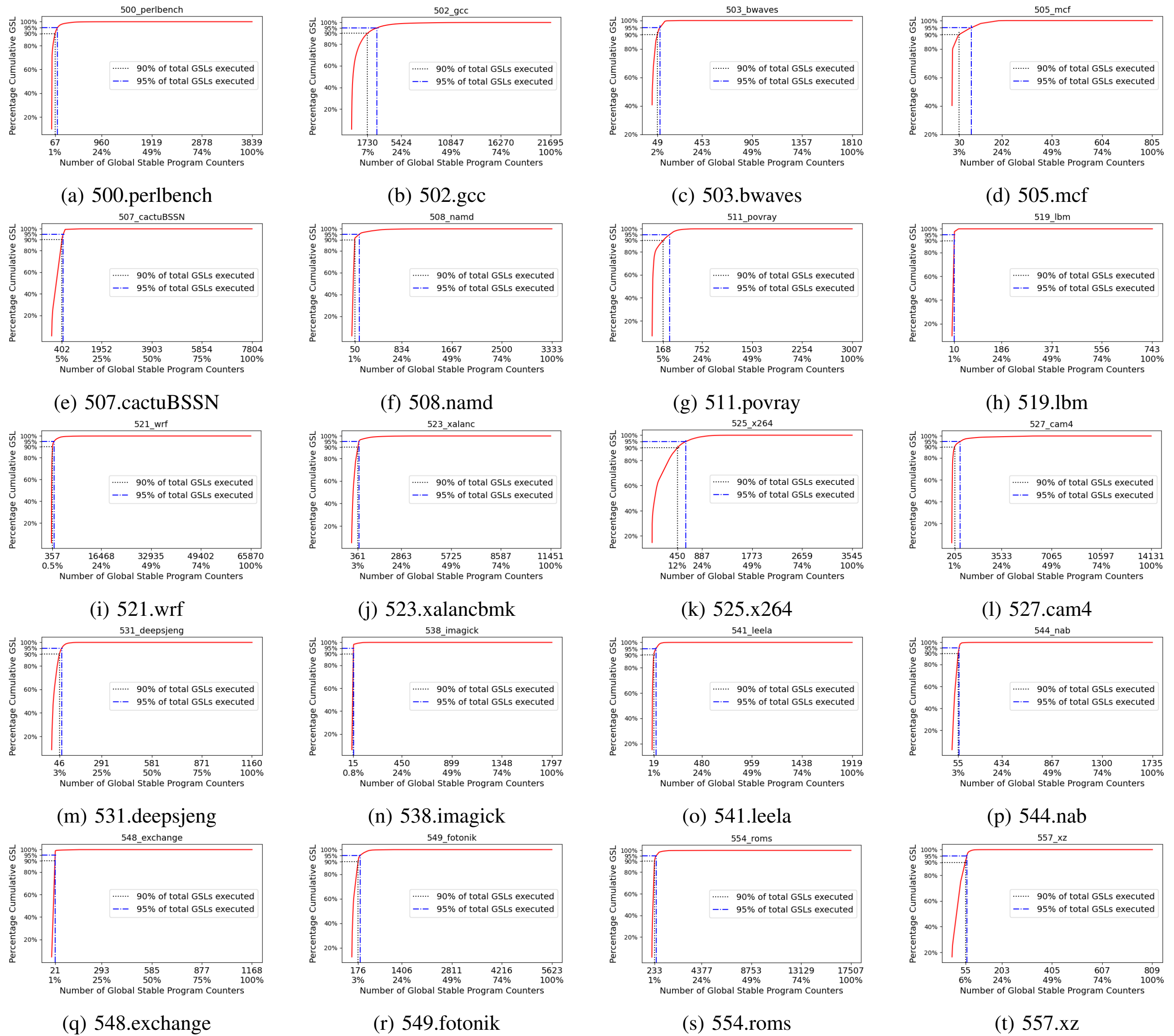
C. Memory Locality in GSLs

Figure 2 shows a high percentage of static GSLs, but a lower dynamic GSL percentage in comparison. We explore whether all GSLs affect performance equally. Is eliminating *all* GSLs necessary, or do GSLs exhibit *locality*, i.e. a few static GSLs responsible for most redundant dynamic loads? Some GSLs may be invoked billions of times, while other GSLs may be invoked only a handful of times. Prior work has shown high locality in the instructions of SPEC95 [27] and SPEC2006 [28] programs. If GSLs too exhibit high locality, we can develop efficient microarchitectural structures targeting only the dominant GSLs.

TABLE III: Static load PCs covering 90%-95% of dynamic GSLs. We report the number of such PCs and what percentage of dynamic GSLs they account for. Programs showing very high locality (90% GSLs captured in 30 PCs or less) are highlighted in green

Program Name	No. of PCs for 90% dynamic GSLs	Percentage out of all GSL PCs	Percentage out of all Load PCs	No. of PCs for 95% dynamic GSLs	Percentage out of all GSL PCs	Percentage out of all Load PCs
500.perlbench	67	1.75%	0.20%	112	2.92%	0.33%
502.gcc	1730	7.97%	1.05%	2745	12.65%	1.66%
503.bwaves	49	2.71%	0.67%	73	4.03%	0.99%
505.mcf	30	3.73%	0.66%	79	9.81%	1.73%
507.cactuBSSN	402	5.15%	1.10%	451	5.78%	1.24%
508.namd	50	1.5%	0.36%	123	3.69%	0.89%
511.povray	168	5.59%	1.08%	264	8.78%	1.70%
519.lbm	10	1.35%	0.25%	10	1.35%	0.25%
521.wrf	357	0.54%	0.17%	842	1.28%	0.4%
523.xalancbmk	361	3.15%	0.78%	417	3.64%	0.91%
525.x264	450	12.69%	3.04%	600	16.93%	4.06%
527.cam4	205	1.45%	0.29%	593	4.2%	0.83%
531.deepsjeng	46	3.97%	0.63%	59	5.09%	0.81%
538.imagick	15	0.83%	0.17%	16	0.89%	0.18%
541.leela	19	0.99%	0.19%	38	1.98%	0.38%
544.nab	55	3.17%	0.79%	62	3.57%	0.89%
548.exchange2	21	1.8%	0.16%	22	1.88%	0.17%
549.fotonik3d	176	3.13%	1.04%	230	4.09%	1.35%
554.roms	233	1.33%	0.67%	375	2.14%	1.07%
557.xz	55	6.8%	0.87%	61	7.54%	0.96%
Average SPEC		**4.27%**	**0.53%**		**5.23%**	**0.64%**

We sort static GSL Program Counters (PCs) in the SPEC benchmarks in descending order by their dynamic instance count, and plot the number of static PCs (x-axis) against cumulative dynamic GSL percentage (y-axis) in Figure 5. We



(a) 500.perlbench (b) 502.gcc (c) 503.bwaves (d) 505.mcf (e) 507.cactuBSSN (f) 508.namd (g) 511.povray (h) 519.lbm (i) 521.wrf (j) 523.xalancbmk (k) 525.x264 (l) 527.cam4 (m) 531.deepsjeng (n) 538.imagick (o) 541.leela (p) 544.nab (q) 548.exchange (r) 549.fotonik (s) 554.roms (t) 557.xz

Fig. 5: Exploring GSL locality in SPEC benchmarks: we sort static GSL PCs by their dynamic count, and plot their cumulative contribution to total dynamic GSLs till that point against the number of PCs till that point. Within 10% of the global stable PCs, the executed GSLs reach near 100% of all the dynamic GSLs for most of the benchmarks.

observe an interesting phenomenon, that most of the dynamic GSLs are contained in a few static PCs. A sharp rise in the red curve (e.g., *519.lbm*) shows strong locality, while a slower rise (e.g., *502.gcc*) indicates weaker locality—yet *502.gcc* still reaches 90% dynamic GSLs with about 8% static GSLs. Blue dashed and dotted lines correspond to 90% and 95% dynamic GSLs, respectively.

Table III lists the number of PCs needed to attain 90% and 95% of the total dynamic GSLs. It is worth noting that in *lbm*, just 10 PCs cover 90% of the dynamic invocations of all GSLs, and in *imagick*, 15 PCs cover 90% of dynamic GSLs. On average, 4.27% (5.23%) PCs out of all GSL PCs cover 90% (95%) of all the dynamic GSLs respectively. Additionally, it can be observed that 0.53% (0.64%) of all loads are sufficient to cover 90% (95%) of dynamic GSLs. The locality of GSLs is relatively weak in *x264* and *gcc*, however, GSLs are fairly localized even in those programs (with only 12.7% and 7.97% GSL PCs needed to reach 90% of the dynamic GSLs).

It is worth noting that in *519.lbm* and *538.imagick*, just 10 and 15 PCs respectively cover 90% of all dynamic GSLs. On average, 4.27% (5.23%) PCs out of all GSL PCs cover 90% (95%) of all dynamic GSLs. Furthermore, 0.53% (0.64%) of all load PCs are sufficient to cover 90% (95%) of dynamic GSLs. The locality of GSLs is relatively weak in *525.x264* and *502.gcc*, however, GSLs are fairly localized even in those programs (with only 12.7% and 7.97% GSL PCs respectively needed to reach 90% of the dynamic GSLs).

In summary, most of the dynamic GSLs are localized around

a few PCs. Prior work [1] created a complex microarchitectural solution to identify and eliminate *all* the GSLs. However, we can instead exploit this locality, and gain similar performance gains by targeting just these dominant GSLs. If a compiler can identify these dominant GSL PCs, it might be possible to design a simple microarchitectural mechanism to exploit this locality. A compiler can provide hints for these likely stable loads and the microarchitecture can perform optimized processing of those loads.

V. Compiler Hints for Likely Stable Loads

In the previous section, we observed the locality patterns in GSLs, that around 5% of the static GSLs make up for more than 90% of the dynamic GSLs. This motivates us to investigate what causes those dominant 5% static GSLs, and whether the compiler can identify them as likely stable loads. After studying the disassemblies of all SPEC programs, we find a few key patterns in the dominant static GSLs that can be identified at compile time.

A. Reading the Canaries Stored on Stack

A canary [29] is a security mechanism where the compiler places a known value (canary) between the function's local variables and its return address. Before returning from the function, the canary value is compared with the original value. Buffer overflows overwrite this canary, allowing attacks to be detected. The canary is placed at a fixed stack location and stays constant within the function's scope, making its load a GSL.

```
000000000041bf50 <_ZN9FastBoard11no_eye_fillEi>:
   ...
41bf64: mov  %fs:0x28,%rax  ;Load Canary to %rax;     4.74B dynamic GSLs
   ...
41c01f: sub  %fs:0x28,%rdx  ;Check if Canary Changed; 4.74B dynamic GSLs
```

Fig. 6: Canary in *541.leela_r* is a significant cause of GSLs

Figure 6 shows parts of the disassembly of a function in *541.leela*. The canary value is first loaded from the stack onto %rax using a "`mov`" , and before returning from the function the canary value in %rax is compared against the current canary value on the stack using a "`sub`". In both these loads, the address and the value remain constant throughout the scope of the program, making them GSLs. Just 6 such static loads in *541.leela* account for 68% of all dynamic GSLs.

We observed GSLs due to canaries present in large numbers in *511.povray* and *502.gcc* as well. We disabled canaries using gcc's compile-time flag `-fno-stack-protector`, and show the decrease in GSLs in table IV. Looking at reductions dynamic GSLs, we can infer how much the canaries contributed to the GSLs. An interesting observation is that *541.leela* shows 88% reduction (5.10% to 0.57%) in dynamic GSLs solely by eliminating canaries. While disabling canaries isn't recommended for security, the results highlight opportunities to identify GSLs at compile-time.

TABLE IV: Comparing GSLs with and without canaries. Canaries are disabled by applying the compiler directive " `-fno-stack-protector` ".

Program	Canaries	Instr	dynamic loads	dynamic GSLs	% dynamic GSLs
541.leela_r	Present	2.3T	613B	31B	5.10%
	Absent	2.2T	555B	3.2B	0.57%
511.povray_r	Present	3.3T	1.2T	131B	10.94%
	Absent	3.2T	1.1T	82B	7.32%
502.gcc_r	Present	218B	57.7B	3.5B	6.07%
	Absent	215B	55.9B	2.9B	4.12%

B. Fetching Constant Addresses from Global Offset Table

The Global Offset Table (GOT) stores addresses of functions and variables in shared libraries. The Procedure Linkage Table (PLT) uses these entries to resolve and call functions at runtime, allowing programs to use shared libraries without needing their exact addresses at compile time. Figure 7 depicts

```
0000000000401020 <.plt>:
   ...
401088: jmp *0x5acff2(%rip) # 9ae080 <_GLOBAL_OFFSET_TABLE_+0x80> ;215M dynamic GSLs
   ...
4010f0: jmp *0x5acff2(%rip) # 9ae0e8 <_GLOBAL_OFFSET_TABLE_+0xe8> ;256M dynamic GSLs
   ...
401178: jmp *0x5acff2(%rip) # 9ae170 <_GLOBAL_OFFSET_TABLE_+0x170>  ;667M dynamic GSLs
```

Fig. 7: Disassembly of *507.cactuBSSN_r*, loading constant addresses from GOT leads to the loads being global stable.

one such example of this scenario, from the disassembly of *507.cactuBSSN*. The instructions are fetching the address values of functions in shared libraries from the GOT. These addresses remain constant throughout the program execution, and therefore these loads are globally stable. These type of GSLs are seen in almost all the SPEC programs. We have therefore identified another class of instructions which can be identified by the compiler to be likely stable.

C. Reading Constants from .data Section

The .data section contains the initialized global and static variables. In several cases, the variable value remains the same throughout the scope of the program execution. Thus all loads for that variable are global stable. Figure 8 shows a section from the disassembly of *519.lbm*. We see that each of those instructions loads from a fixed offset from *_IO_stdin_used*, which is a symbol in glibc. *_IO_stdin_used* is placed in the .data section, and all these instructions are loading some global variable whose value does not change. Therefore, these instructions become global stable. An interesting thing to note here is that such 10 instructions account for 35B out of the total 36B dynamic GSLs, which is 97% of the total GSLs. This observation further motivates us that we can identify a large fraction of GSLs at compile time itself, utilize hardware-software co-design, and create simple microarchitectural mechanisms that work in conjunction with compiler hints.

VI. A Compiler-Hardware Co-Design

Motivated by the opportunity of the compiler to identify key GSLs, we propose a compiler-assisted microarchitectural

```
0000000000402a60 <LBM_performStreamCollideTRT>:
   ...
402ae4: vmovsd 0xb668c(%rip),%xmm9        # 4b9178 <_IO_stdin_used+0x178>; 3.5B GSLs
   ...
402b4f: vmovsd 0xb6729(%rip),%xmm7        # 4b9280 <_IO_stdin_used+0x280>; 3.5B GSLs
   ...
402b57: vmulsd 0xb6611(%rip),%xmm8,%xmm8  # 4b9170 <_IO_stdin_used+0x170>; 3.5B GSLs
```

Fig. 8: GSLs in dissassembly of *519.lbm_r*, loading constant variables from the .data section.

solution. Prior work [30]–[33] has shown that it is indeed possible to statically eliminate such redundant loads at compile time. Our idea is that the compiler will identify and annotate these dominant static GSLs. We then add a tiny fast cache in the microarchitecture dedicated to store the data from the GSLs.

A. ISA Changes: The LS version of Instructions

We propose a new class of instructions, namely Likely Stable (LS) instructions, that the compiler can use to annotate potential GSLs. Likely stable instructions are defined as any instruction involving a memory load operation which the compiler thinks will fetch the same value from the same address. Such instructions would be replaced by their likely stable versions. For instance, in the instruction "`ADD 0x8(rbp),rax`" if compiler determines that the source address 0x8(rbp) will evaluate to a constant address, and the value at that address would be a constant too, it replaces this `ADD` with `ADDLS` instruction. Table V shows some examples of this new class of likely stable instructions. These new instructions are passed as a hint to the microarchitecture.

TABLE V: Examples of Likely Stable (LS) version of instructions, defined for instructions that contain a load operation.

Instruction	Example
`MOV-LS`	eg: `MOVLS 0x4(esp),eax`. 0x4(esp) always evaluates to the same address and m[esp+0x4] always returns the same value
`CMP-LS`	eg: `CMP eax, 0x8(rsp)` reads from m[rsp+0x8]. If identified as a constant load, the compiler converts it to: `CMPLS eax, 0x8(rsp)`
`ADD-LS`	eg: `ADDLS 0x8(rbp),eax`. Arithmetic also involves load operations. In this case, rbp+0x8 and m[rbp+0x8] remain constant.

B. Compiler Changes

We discuss the required compiler changes to support the likely stable annotations. In the front-end during the parsing phase (lexical & syntax analysis), for each line of code the compiler looks out for certain keywords or patterns to check for likely stable instructions. For instance, a global variable or a libc call (printf) can be detected by the compiler to be likely stable instructions. During lexical analysis when the source code is broken down into tokens, for these special lines of code an additional token named LS (likely stable) is added. During semantic analysis, the compiler walks through the parsed tree and checks if the variable value may be changed along the path. This gives us more confidence in the LS token indeed being global stable. The parsed tree generated after semantic analysis has the LS annotations in it which get carried on till the IR phase. Finally, during the code generation phase, the assembly code is created with our new LS instruction variant of the corresponding instruction as shown in table V.

There are other global stable cases such as canary insertions which occur during the code generation phase. Therefore, the code generation phase is also modified so that for operations such as canary insertions, the assembly is created with the LS variant of that instruction.

C. A Small Fast L0 Cache for GSLs

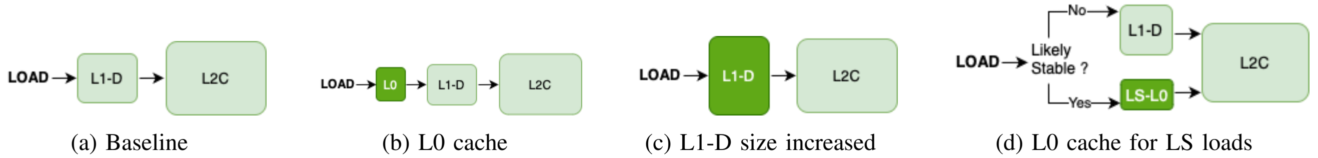
Now that we have special instructions for likely stable loads, we add a small and fast L0 cache that would store only the likely stable loads. Prior research [34]–[36] has explored adding an L0 cache to the memory hierarchy. The idea is to exploit the reduced energy and access latency of L0 compared to L1. In fact, Intel has an L0 cache in it's recent Lunar Lake processor [37]. The drawback of L0 is the increased miss rate due to the small size. Prior solutions [34]–[36], [38] proposed complex prediction and management schemes to reduce the miss rate. In our case however, the miss rate would be low because we are storing only the compiler marked stable loads in the L0 cache. As we saw in section IV-C, storing only the top 5% of the GSL PCs is enough give us significant improvement. Therefore, even a small L0 cache can give us good hit rates.

The added L0 cache to store the likely stable (LS) loads, is called the LS-L0 cache. Figure 9d shows the LS-L0 cache integrated in the memory hierarchy. Any load that is marked as LS sends its read request to the LS-L0 cache. If there is a read miss in the LS-L0 cache, the data is fetched from L2 cache. All the other loads that are not marked as LS follow the regular memory hierarchy starting from L1-D. Since there are two parallel datapaths from the core to L1-D and LS-L0, it is crucial to handle coherence between L1-D and LS-L0. We add LS-L0 in the existing coherence protocol at L1 level, and LS-L0 maintains its coherence states in a separate tag array.

It is important to note that even if the compiler misidentifies an LS load, it may degrade the performance but not affect the correctness in execution. For instance, consider a load which was incorrectly marked by the compiler as LS load. Since the value is no longer globally constant, another instruction may request a write to that address while that load value is present in the LS-L0 cache. The new value now occupies the L1-D cache, and the coherency protocol invalidates the corresponding block in the LS-L0 cache.

VII. Evaluation and Results

In this section, we evaluate the speedup and energy savings achieved by utilizing the LS-L0 cache (figure 9d) compared to the baseline. Figure 9a is our baseline, with only L1-D cache above L2 level. We used the ChampSim [12] simulator for our experiments. We annotate the traces of SPEC programs marking the loads which are GSLs, and use those traces for our experiments with Champsim. Energy and access time estimates of added microarchitectural structures are estimated using CACTI [39]. Table VI shows the microarchitectural parameters used in our simulations. We also compare our results with Constable [1], the state-of-the-art microarchitectural solution to eliminate GSLs.



(a) Baseline (b) L0 cache (c) L1-D size increased (d) L0 cache for LS loads

Fig. 9: The 4 cache configurations used in our evaluations. 9a is our baseline, and the modifications over baseline are colored in dark green. 9b adds a small and fast L0 cache in the memory hierarchy, with all loads now passing through L0 to go to L1-D.9c increases the size of L1-D cache by the size of L0 cache. 9d is our proposed LS-L0 cache dedicated to store likely stable loads. The likely stable (LS) loads are sent to LS-L0 while all other load requests go to L1-D cache.

TABLE VI: MicroArchitectural Configuration

Basic	4GHz, single-core, out-of-order, block size:64, page size:4096
Core	6-wide fetch, 6-wide decode, 6-wide dispatch, 4-wide execute, load queue width: 2, store queue width: 2, retire width: 5, fetch buffer size: 64, decode buffer size: 32, dispatch buffer size: 32, ROB size: 352, load queue size: 128, store queue size: 72, scheduler size: 128, bimodal branch predictor
Physical Memory	3.2GHz, channels:1, ranks:1, banks:8, rows:65536, columns:128, row size:8
LLC	2MB 16-way set associative, MSHR size:64, latency:20 cycles, Read Energy: 282.5pJ, Write Energy: 307.1pJ
L2 Cache	512KB 8-way set associative, MSHR size:32, latency:10 cycles, Read Energy: 127.1pJ, Write Energy: 172.9pJ
L1-I Cache	32KB 8-way set associative, MSHR size:8, latency:5 cycles, Read Energy: 81.2pJ, Write Energy: 81.6pJ
L1-D Cache	48KB, 8-way set associative, MSHR size:16, latency:6 cycles, Read Energy: 82.3pJ, Write Energy: 84.1pJ
Large L1-D	54KB, 8-way set associative, MSHR size:16, latency:6 cycles, Read Energy: 82.7pJ, Write Energy: 85.1pJ
L0 Cache	6KB Direct Mapped, MSHR size:1 , latency:1 cycle , Read Energy: 8.2pJ, Write Energy: 10.9pJ
LS-L0 Cache	6KB Direct Mapped, MSHR size:1 , latency:1 cycle , Read Energy: 8.2pJ, Write Energy: 10.9pJ

A. Performance

We evaluate the cache hits and speedup of our design with LS-L0 cache (figure 9d) and see the improvement over baseline. We also compare the performance with two other microarchitectural designs as shown in 9. Figure 9b integrates a small and fast L0 cache in the cache hierarchy. Figure 9c increases the size of the baseline's L1-D cache by the size of L0 cache in order to have more storage and increase the chance of a GSL hit. The core, physical memory, LLC, L2, and L1-I cache parameters are the same across the 4 designs.

*1) **Cache Hits**:* Table VII presents the hit ratios of the caches in the four designs. Using a larger L1-D does not bring any change in the hit ratio compared to baseline, because when the cache indexes change by a number that is not a power of two, the mapping does not change considerably. In the design with the L0 cache, all the loads going to L1-D are now passing through L0. Due to its small size, it cannot handle the entire traffic and achieves a poor hit rate (geomean 0.6380). Furthermore, the hit rate of L1-D is also drastically worsened. This shows that simply adding a fast cache cannot provide benefits, we also need a smart data management policy. LS-L0 exploits the compiler annotated LS loads to smartly navigate only the LS loads to LS-L0. Since only the LS-loads are stored in the LS-L0 cache, the amount of traffic coming in is considerably lower than L0, and the data in LS-L0 would not frequently get evicted. Therefore, it achieves a high hit ratio (geomean 0.9244), and also keeps the hit ratio of L1-D similar to baseline.

TABLE VII: Cache Hit Ratios for the four designs

	Baseline (fig. 9a)	**Large L1-D (fig. 9c)**	**L0 Cache (fig. 9b)**		**LS-L0 Cache (fig. 9d)**	
Program	**L1-D Hit Ratio**	**L1-D Hit Ratio**	**L1-D Hit Ratio**	**L0 Hit Ratio**	**L1-D Hit Ratio**	**LS-L0 Hit Ratio**
500.perlbench_r	0.9891	0.9891	0.9478	0.8525	0.9875	0.9512
502.gcc_r	0.9253	0.9253	0.7337	0.7754	0.9132	0.8218
503.bwaves_r	0.9104	0.9104	0.8036	0.4996	0.9029	0.9809
505.mcf_r	0.8776	0.8776	0.576	0.7673	0.8698	0.9952
507.cactuBSSN_r	0.8334	0.8334	0.535	0.6527	0.8181	0.9148
508.namd_r	0.9721	0.9721	0.8518	0.8598	0.9711	0.9981
511.povray_r	0.9774	0.9774	0.9203	0.7867	0.9827	0.9189
519.lbm_r	0.7612	0.7612	0.5704	0.7170	0.7141	0.8516
523.xalanc_r	0.7970	0.7970	0.4947	0.7428	0.7699	0.9678
525.x264_r	0.9940	0.9940	0.951	0.9241	0.9939	0.9129
527.cam4_r	0.9593	0.9593	0.8374	0.8199	0.9435	0.8874
531.deepsjeng_r	0.9952	0.9952	0.9759	0.8559	0.9951	0.876
538.imagick_r	0.6426	0.6426	0.0239	0.6285	0.6277	0.8886
541.leela_r	0.9826	0.9826	0.8963	0.8896	0.9795	0.9604
544.nab_r	0.9530	0.9530	0.6952	0.8777	0.8884	0.9998
548.exchange2_r	0.9999	0.9999	0.9999	0.9337	0.9999	0.9712
549.fotonik3d_r	0.8924	0.8924	0.6492	0.7156	0.8857	0.9405
554.roms_r	0.9446	0.9446	0.827	0.7248	0.9081	0.8261
557.xz_r	0.9560	0.9560	0.8291	0.8284	0.9504	0.9328
GeoMean SPECint	0.9439	0.9439	0.8015	0.8385	0.9368	0.9306
GeoMean SPECfloat	0.8778	0.8778	0.5196	0.7193	0.8567	0.9189
Total GeoMean	0.9085	0.9085	0.6380	0.7735	0.8937	0.9244

TABLE VIII: IPC and Speedup for the four designs

	Baseline (fig. 9a)	**Large L1-D (fig. 9c)**	**L0 Cache (fig. 9b)**		**LS-L0 Cache (fig. 9d)**	
Program	**IPC**	**IPC**	**IPC**	**Speedup**	**IPC**	**Speedup**
500.perlbench_r	0.9179	0.9179	0.9350	1.86%	0.9470	3.18%
502.gcc_r	0.3052	0.3052	0.2212	-27.54%	0.3077	0.83%
503.bwaves_r	0.4026	0.4026	0.1262	-68.66%	0.4052	0.66%
505.mcf_r	0.3392	0.3392	0.1569	-53.76%	0.3393	0.01%
507.cactuBSSN_r	0.4748	0.4748	0.0663	-86.02%	0.5163	8.74%
508.namd_r	1.9168	1.9168	1.5145	-20.99%	1.9955	4.10%
511.povray_r	1.1850	1.1850	1.0378	-12.43%	1.2715	7.3%
519.lbm_r	0.4930	0.4930	0.1119	-77.32%	0.5439	10.33%
523.xalanc_r	0.2302	0.2302	0.1517	-34.11%	0.2315	0.57%
525.x264_r	2.4393	2.4393	2.0918	-14.25%	2.4531	0.57%
527.cam4_r	0.9236	0.9236	0.6024	-34.78%	1.0181	10.23%
531.deepsjeng_r	1.0284	1.0284	1.0605	3.11%	1.0333	0.48%
538.imagick_r	2.3538	2.3538	1.6688	-29.1%	2.3576	0.16%
541.leela_r	1.0009	1.0009	1.0697	6.88%	1.0273	2.64%
544.nab_r	0.8345	0.8345	0.6877	-17.6%	1.0549	26.4%
548.exchange2_r	0.9286	0.9286	1.0137	9.16%	0.9374	0.95%
549.fotonik3d_r	0.5427	0.5427	0.1300	-76.05%	0.5445	0.32%
554.roms_r	0.7382	0.7382	0.2948	-60.06%	0.8005	8.43%
557.xz_r	0.6561	0.6561	0.4748	-27.64%	0.6446	-1.76%
GeoMean SPECint		No Speedup		-41.15%		**0.82%**
GeoMean SPECfloat		No Speedup		-17.98%		**7.43%**
Total GeoMean		No Speedup		-56.35%		**4.24%**

*2) **Performance Improvement**:* We measure the Instructions per cycle (IPC) in the four scenarios, and present the speedup as a percentage increase in IPC in table VIII. A larger L1-D did not improve the hit ratio as we saw in section VII-A1, and therefore shows no speedup. The design with L0 integrated in the memory hierarchy results in a high slowdown of 56.35%. Since L0 has a poor hit ratio, almost all the loads go to L1-D with an additional latency of L0. However, LS-L0 achieves a speedup of 4.24% over the baseline due to the high hit ratio of LS-L0 coupled and its low access latency.

The *544.nab* benchmark shows the maximum improvement of 26.4%. This can be attributed to the fact that *544.nab* has the highest percentage of dynamic GSLs among all SPEC Programs as shown in figure 2. Thus, a high number of LS loads are routed through the fast LS-L0 cache, thereby reducing memory latency.

Performance with a small L0 cache depends on data mapping policies, often requiring complex policies [34]–[36], [38], [40] to mitigate capacity or conflict misses and achieve speedup. Prior work such as using history based prediction to choose between L0 and L1 [34] resulted in 0.7% performance degradation, while promoting cache lines to L0 on only on L1 hits [35] negatively impacted performance by 0-5%. However, our policy of mapping the GSLs to LS-L0 achieves a 4.24% speedup. Previous state-of-the-art work [1] achieved a speedup of around 1% and 5% for the SPEC Int and Float benchmarks, respectively. Using the LS-L0 cache, we achieve a geomean speedup of 0.82% and 7.43% for SPEC Int and Float, respectively, with a simpler architecture.

B. Energy

Prior studies [34]–[36] on L0 caches have demonstrated its ability to lower power consumption. The LS-L0 cache not only has low power requirements but also decreases the number of accesses to L1-D, thereby decreasing the overall energy of the levels above L2. Furthermore, the high hit ratio of LS-L0 cache makes sure that we do not have to access L2C frequently, and saves energy on that front as well. We compute the total energy for cache accesses (LS-L0,L1-D,L2C,LCC) and compare it against the baseline's total cache access energy (L1-D,L2C,LLC). Figure 10 presents the percentage energy reductions for the SPEC programs.

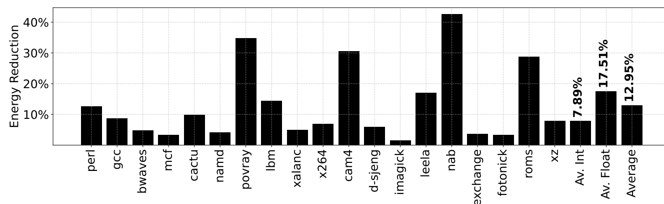


Fig. 10: We compute the reduction in cache access energy of combined LS-L0+L1 accesses compared to baseline's L1

The *544.nab* benchmark shows the highest energy reduction of 42.6%. This is despite the fact that the hit ratio of L1-D drops from 0.95 to 0.88. *544.nab* has the highest percentage of dynamic GSLs (as shown in figure 2), which led to more than 50% of the loads accessing the LS-L0 cache. That more than compensates for the increased energy consumption due to L1-D misses. The number of accesses to LS-L0 plays a key role in energy reduction. For instance *505.mcf* has a high LS-L0 hit ratio of 0.99, but still shows a only slight reduction of 3.93%. This is because in *505.mcf* has less than 1% dynamic GSLs, and not many loads go through the LS-L0 cache. Therefore, programs with large number of dynamic GSLs will result in lot of loads going through the LS-L0 and thereby save a lot of energy.

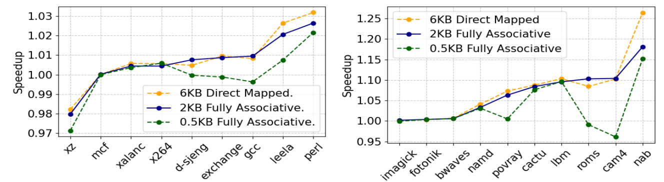
The average energy reduction in SPEC Int and Float programs are 7.89% and 17.51% respectively. The higher reduction in SPEC Float is due to higher number of dynamic GSLs than SPEC Int. Constable achieved a power reduction of 9.1% for the L1-D cache while our LS-L0 cache can get a power reduction of 12.95%.

C. Sensitivity Study

So far we used a 6KB direct mapped LS-L0 cache. This section compares its performance against two other configurations – a 2KB fully associative and a 0.5KB fully associative – to assess size and associativity effects. Figure 11 compares the speedup in all SPEC programs and table IX lists the geomean speedups for the three caches. The 6KB direct mapped LS-L0 achieves the best speedup due to its lower latency and larger size. The low latency compensates for any increased conflicts due to reduced associativity. On the other hand, a 0.5KB fully associative cache is too small to avoid the capacity misses and performs poorly.

TABLE IX: Speedups with Different L1-Fast Cache Configurations. Block Size for all Caches is 64B

LS-L0: Size & Associativity	Latency	GeoMean Speedup SPECrate Int	GeoMean Speedup SPECrate Float	GeoMean Speedup SPECrate
6KB Direct Mapped	1-cycle	0.82%	7.43%	4.24%
2KB Fully Associative	2-cycle	0.67%	6.60%	3.75%
0.5KB Fully Associative	1-cycle	-0.01%	3.07%	1.56%



(a) SPECrate Int

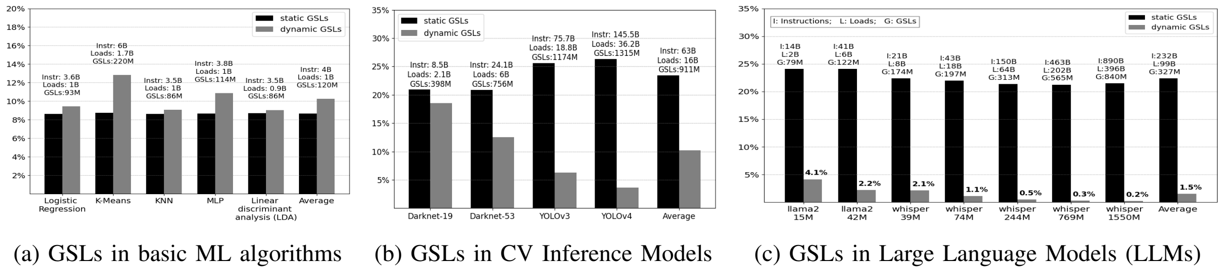
(b) SPECrate Float

Fig. 11: Comparing the speedups with different configurations of LS-L0 cache. The block size is 64B for all configurations. On the x-axis, the programs are sorted by their speedup with the 2KB fully associative LS-L0 cache.

VIII. Exploring GSLs in ML Workloads

Machine learning (ML) applications have become ubiquitous in today's world. Prior study of GSLs [1] did not include any AI/ML workload in their study. In this section, we investigate the presence of global stable loads in machine learning workloads. In particular, we focus on ML inference, since training is predominantly done on GPUs rather than CPUs. We start with small ML models, move to larger CNNs, and finally explore some large language models.

Fundamental ML Algorithms: Figure 12a shows the static and dynamic GSLs in the models of Logistic Regression, K-Means, K-Nearest Neighbors (KNN), Multilayer Perceptron (MLP) and Linear Discriminant Analysis (LDA) . We get an



(a) GSLs in basic ML algorithms (b) GSLs in CV Inference Models (c) GSLs in Large Language Models (LLMs)

Fig. 12: Studying GSLs in various ML workloads shows that dynamic GSLs decrease as the model size increases.

average of 10% dynamic GSLs and 8.7% static GSLs. These numbers are considerably than what prior [1] work observed in client, server, and enterprise workloads (35%-55%). However, these ML models are quite small and we explore how GSLs change as we increase the model size.

CNNs for Object Detection: Real time object detection algorithms are becoming increasingly prevalent across diverse fields such as autonomous driving, security, and retail analytics. In this section we focus on CNN inference models for object detection. Figure 12b shows the dynamic GSLs in CNN workloads. Darknet-19 and Darknet-53 have 19 and 53 convolutional layers respectively. YOLOv3 has 53 convulutional layers while YOLOv4 has 161 layers including 80 convolutional layers. We see that the CNN workloads show an average 23.4% static GSLs and 10.2% dynamic GSLs. Furthermore, as the model size continues to increase from Darknet-19 to YOLOv4, we see a consistent decrease in the percentage of dynamic GSLs from 18. 5% to 3. 6%.

Large Language Models: Generative-AI, like ChatGPT, is increasingly integrated into workflows, making Large Language Model (LLM) inference a substantial CPU workload. In this section, we investigate GSLs during LLM-inference. We study two architectures, baby-llama2 [13], [14] emulating Meta's llama2 model, and OpenAI's whisper [15]–[17] which is a speech recognition model. Figure 12c shows a consistent decrease in the percentage of dynamic GSLs as the size of the model increases. Even the model with highest percentage of dynamic GSLs (baby-llama2 with 15M parameters) has only 4.1% dynamic GSLs. The percentage dynamic GSLs decreases to as low as 0.5% for the whisper model with 244M parameters and goes down to 0. 2% with the model having 1.5B parameters.

Summary: We observe that as the model size increases, the percentage of dynamic GSLs considerably decreases, going below even 1% for LLM workloads. This is because in these large models with millions or billions of different weights, the likelihood of repeatedly fetching the same value from the same address is very low. Thus, as our workload paradigm is shifting to larger and larger models, we may need to worry less about eliminating GSLs.

IX. RELATED WORK

Redundant computation is documented in significant prior work [41]–[45] as well. Sodani et al. [46] found that up to 80% of dynamic instructions repeat with the same inputs and outputs. A mechanism for recording and reusing results of instructions when their operands remain the same across executions – effectively implementing memoization at the instruction level in hardware. There have been several other memoization based [47]–[55] techniques to mitigate redundant computation. For example, Costa et al. [56] propose a technique called *dynamic trace memoization* that skips the execution of redundant instructions by reusing their previous values. Moura et al. [57] study trace reuse in arm architectures and propose a memoization based solution. Several value prediction mechanisms [58]–[60] have also been proposed to eliminate this redundant execution. For instance, Lipasti load value predictor [61] eliminates the data fetches by caching memory locations that are predominantly constant. Recently, Bera and Ranganathan et al. [1] had proposed a microarchitecural solution to dynamically identify, track and eliminate GSLs.

The key drawback in prior solutions is the complexity and hardware overhead of memoization and tracking. Previous works did not perform any locality analysis as we did in section IV-C. They plan to eliminate all the redundancies, while we plan on targeting just the dominant ones. We use a simpler hardware-software co-designed solution compared to a complex microarchitectural design.

X. CONCLUSION

We analyze Global Stable Loads (GSLs) reported in prior work, and find that they are not an artifact of a specific architecture or compiler, and compiler optimizations alone cannot eliminate them. We also study ML workloads and find that GSLs diminish as models grow in size. We do a locality study and find that only about 4%-5% of GSL PCs are responsible for 90%-95% of all dynamic GSLs. We identify the reasons for these dominant GSLs, and propose the compiler to hint them as likely stable. Leveraging this locality and compiler hints, we add a small and fast L0 cache to handle these likely stable loads. We achieve similar speedup and energy reduction as state-of-the-art solution, while using less hardware.

ACKNOWLEDGMENTS

We thank Molly O'Neil for her valuable insights and discussions during the preparation of this paper. We also thank the NVIDIA Applied Research Accelerator Program Grant.

REFERENCES

[1] R. Bera, A. Ranganathan, J. Rakshit, S. Mahto, A. V. Nori, J. Gaur, A. Olgun, K. Kanellopoulos, M. Sadrosadati, S. Subramoney, and O. Mutlu, "Constable: Improving performance and power efficiency by safely eliminating load instruction execution," in *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ser. ISCA '24. IEEE Press, 2025, p. 88–102. [Online]. Available: https://doi.org/10.1109/ISCA59077.2024.00017

[2] GNU-Project, "GCC Optimization Flags," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, 2024.

[3] O. Corporation, "x86 assembly language reference manual," https://docs.oracle.com/cd/E53394_01/pdf/E54851.pdf, 2013, accessed: 2025-02-28.

[4] CMU-SAFARI, "Load-inspector," 2024. [Online]. Available: https://github.com/CMU-SAFARI/Load-Inspector

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[6] Standard Performance Evaluation Corporation, "Spec cpu 2017 benchmark suite," 2017. [Online]. Available: https://www.spec.org/cpu2017/

[7] GNU Project, "GNU Compiler Collection (GCC)," https://gcc.gnu.org/, 1987, accessed: February 2025.

[8] LLVM Project, "Clang: a C language family frontend for LLVM," https://clang.llvm.org/, 2008, accessed: February 2025.

[9] AMD, "Md epyc 7742 processor." [Online]. Available: https://www.amd.com/en/products/cpu/amd-epyc-7742

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[11] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[12] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The Championship Simulator: Architectural Simulation for Education and Competition," 2022.

[13] A. Karpathy, "llama2.c: A minimal implementation of llama-2 in c," https://github.com/karpathy/llama2.c, 2024.

[14] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023. [Online]. Available: https://arxiv.org/abs/2307.09288

[15] G. Gerganov, "whisper.cpp: High-performance whisper model inference in c/c++," https://github.com/ggerganov/whisper.cpp, 2024.

[16] OpenAI, "Whisper: Robust speech recognition via large-scale weak supervision," 2022, accessed: 2024-11-23. [Online]. Available: https://github.com/openai/whisper

[17] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," 2022. [Online]. Available: https://arxiv.org/abs/2212.04356

[18] A. Bochkovskiy, "Darknet: Open source neural networks in c," https://github.com/AlexeyAB/darknet, 2024.

[19] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[20] A. Bochkovskiy, C. Wang, and H. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *CoRR*, vol. abs/2004.10934, 2020. [Online]. Available: https://arxiv.org/abs/2004.10934

[21] E. Lindernoren, "Napkinml: A simple machine learning library," https://github.com/eriklindernoren/NapkinML, 2024, accessed: 2024-11-22.

[22] K. Hayen, "Nuitka: Python compiler with full language support," https://github.com/Nuitka/Nuitka, 2024.

[23] J.-J. Kim, S.-Y. Lee, S.-M. Moon, and S. Kim, "Comparison of llvm and gcc on the arm platform," in *2010 5th International Conference on Embedded and Multimedia Computing*, 2010, pp. 1–6.

[24] N. Sakib, T. Prabhu, N. Santhi, J. Shalf, and A.-H. A. Badawy, "Comparison of vectorization capabilities of different compilers for x86 and arm cpus," 2025. [Online]. Available: https://arxiv.org/abs/2502.11906

[25] C. Park, M. Han, H. Lee, and S. W. Kim, "Performance comparison of gcc and llvm on the eisc processor," in *2014 International Conference on Electronics, Information and Communications (ICEIC)*, 2014, pp. 1–2.

[26] M. D. Brown, M. Pruett, R. Bigelow, G. Mururu, and S. Pande, "Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, p. 1–30, Oct. 2021. [Online]. Available: http://dx.doi.org/10.1145/3485531

[27] P. Vasudevan, L. K. John, and J. Sabarinathan, " Workload Characterization: Motivation, Goals and Methodology ," in *Workload Characterization, Annual IEEE International Workshop*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 1998, p. 3. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/WWC.1998.809354

[28] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 412–423. [Online]. Available: https://doi.org/10.1145/1250662.1250713

[29] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. USA: USENIX Association, 1998, p. 5.

[30] R. Espasa and M. Fernández, " Link-Time Path-Sensitive Memory Redundancy Elimination ," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Feb. 2004, p. 300. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA.2004.10009

[31] P. Su, S. Wen, H. Yang, M. Chabbi, and X. Liu, "Redundant loads: A software inefficiency indicator," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 982–993.

[32] Y. Wu and Y.-f. Lee, "Comprehensive redundant load elimination for the ia-64 architecture," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, L. Carter and J. Ferrante, Eds. Berlin, Heidelberg: Springer, 2000, vol. 1863.

[33] M. Fernández, R. Espasa, and S. Debray, "Load redundancy elimination on executable code," in *Euro-Par 2001 Parallel Processing*, ser. Lecture Notes in Computer Science, R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, Eds. Berlin, Heidelberg: Springer, 2001, vol. 2150.

[34] W. Tang, A. Veidenbaum, and R. Gupta, "Architectural adaptation for power and performance," in *ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No.01TH8549)*, 2001, pp. 530–534.

[35] N. Duong, T. Kim, D. Zhao, and A. V. Veidenbaum, "Revisiting level-0 caches in embedded processors," in *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '12. New York, NY, USA:

Association for Computing Machinery, 2012, p. 171–180. [Online]. Available: https://doi.org/10.1145/2380403.2380435

[36] W. Tang, A. Veidenbaum, and A. Nicolau, "Reducing power with an l0 instruction cache using history-based prediction," in *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2002, pp. 11–18.

[37] Intel, "Intel lunar lake architecture: Deep dive into lion cove, xe2, and npu4," https://www.anandtech.com/show/21425/intel-lunar-lake-architecture-deep-dive-lion-cove-xe2-and-npu4/2, 2024, accessed: 2024-02-23.

[38] L. John and A. Subramanian, "Design and performance evaluation of a cache assist to implement selective caching," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, 1997, pp. 510–518.

[39] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 694–701.

[40] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 423–432.

[41] C. Molina, A. González, and J. Tubella, "Dynamic removal of redundant computations," in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 474–481. [Online]. Available: https://doi.org/10.1145/305138.305239

[42] S. Richardson, "Exploiting trivial and redundant computation," in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, 1993, pp. 220–227.

[43] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta, "Continuous optimization," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. USA: IEEE Computer Society, 2005, p. 86–97. [Online]. Available: https://doi.org/10.1109/ISCA.2005.19

[44] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. Patel, and S. Lumetta, "Performance characterization of a hardware mechanism for dynamic optimization," *Proceedings of the Annual International Symposium on Microarchitecture*, pp. 16–27, 2001.

[45] A. Gellert, A. Florea, and L. Vintan, "Exploiting selective instruction reuse and value prediction in a superscalar architecture," *J. Syst. Archit.*, vol. 55, no. 3, p. 188–195, Mar. 2009. [Online]. Available: https://doi.org/10.1016/j.sysarc.2008.11.002

[46] A. Sodani and G. S. Sohi, "An empirical analysis of instruction repetition," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII. New York, NY, USA: Association for Computing Machinery, 1998, p. 35–45. [Online]. Available: https://doi.org/10.1145/291069.291016

[47] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating multi-media processing by implementing memoing in multiplication and division units," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII. New York, NY, USA: Association for Computing Machinery, 1998, p. 252–261. [Online]. Available: https://doi.org/10.1145/291069.291056

[48] D. A. Connors and W.-m. W. Hwu, "Compiler-directed dynamic computation reuse: rationale and initial results," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 32. USA: IEEE Computer Society, 1999, p. 158–169.

[49] S. E. Richardson, "Caching function results: Faster arithmetic by avoiding unnecessary computation," USA, Tech. Rep., 1992.

[50] A. Suresh, B. N. Swamy, E. Rohou, and A. Seznec, "Intercepting functions for memoization: A case study using transcendental functions," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2751559

[51] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 194–205. [Online]. Available: https://doi.org/10.1145/264107.264200

[52] A. Gonzalez, J. Tubella, and C. Molina, "Trace-level reuse," in *Proceedings of the 1999 International Conference on Parallel Processing*, 1999, pp. 30–37.

[53] J. Huang and D. J. Lilja, "Extending value reuse to basic blocks with compiler support," *IEEE Trans. Comput.*, vol. 49, no. 4, p. 331–347, Apr. 2000. [Online]. Available: https://doi.org/10.1109/12.844346

[54] G. Zhang and D. Sanchez, "Leveraging caches to accelerate hash tables and memoization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 440–452. [Online]. Available: https://doi.org/10.1145/3352460.3358272

[55] E. Benowitz, M. Ercegovac, and F. Fallah, "Reducing the latency of division operations with partial caching," vol. 2, 12 2002, pp. 1598 – 1602 vol.2.

[56] A. da Costa, F. Franca, and E. Filho, "The dynamic trace memoization reuse technique," in *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, 2000, pp. 92–99.

[57] R. C. De Moura, G. O. Torres, M. L. Pilla, L. L. Pilla, A. T. Da Costa, and F. M. França, "Value reuse potential in arm architectures," in *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2016, pp. 174–181.

[58] S. Bandishte, J. Gaur, Z. Sperber, L. Rappoport, A. Yoaz, and S. Subramoney, "Focused value prediction," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 79–91. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00018

[59] K. Kalaitzidis and A. Seznec, "Leveraging value equality prediction for value speculation," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, Dec. 2021. [Online]. Available: https://doi.org/10.1145/3436821

[60] A. Perais, "Leveraging targeted value prediction to unlock new hardware strength reduction potential," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 792–803. [Online]. Available: https://doi.org/10.1145/3466752.3480050

[61] 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. New York, NY, USA: Association for Computing Machinery, 1996, p. 138–147. [Online]. Available: https://doi.org/10.1145/237090.237173