# Exploiting Dead Value Information

Milo M. Martin, Amir Roth, and Charles N. Fischer
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
{milo, amir, fischer}@cs.wisc.edu

**Abstract**

*We describe Dead Value Information (DVI) and introduce three new optimizations which exploit it. DVI provides assertions that certain register values are dead, meaning they will not be read before being overwritten. The processor can use DVI to track dead registers and dynamically eliminate unnecessary save and restore instructions from the execution stream at procedure calls and context switches. Our results indicate that dynamic saves and restore instances can be reduced by 46% for procedure calls and by 51% for context switches. In addition, save/restore elimination for procedure calls can improve overall performance by up to 5%. DVI also allows the processor manage physical registers to efficiently, reducing the size requirements of the physical register file. When the system clock rate is proportional to the register file cycle time, this optimization can improve performance. All of these optimizations can be supported with only a few new instructions and minimal additional hardware structures.*

## 1 Introduction

Executables provide the processor with a static operational description of the program. In producing an executable, a compiler discovers many facts about the control and data flow properties of the program. However, much of this information is not made explicit in the executable and is left for the hardware to rediscover during execution. In the dynamic instruction sequence below, register r1 is *dead* after I2. The value of a dead register is not needed for continued correct execution, *i.e.* the register will not be read again before it is overwritten.

```
I1:  r1 ←                           (Definition of r1)
I2:  ← r1                           (Use of r1)
     ...
     Many instructions that
     do not use r1
     ...
I3:  r1 ←                           (Redefinition of r1)
```

The interval between instructions I2 and I3 presents many opportunities for optimization. The storage reserved for the value of r1 can be reclaimed anywhere in the interval with no effect on program correctness. The register also need not be preserved across procedure calls or context switches that occur in the interval. Unfortunately, an ordinary processor executing this sequence cannot determine that r1 is dead until it encounters I3, and therefore cannot take advantage of these opportunities.

*Dead Value Information* (DVI) provides assertions about *future* register usage in programs. By inserting a DVI annotation after I2, declaring the value in register r1 dead, the compiler allows the processor to track this information dynamically. In this paper we present and evaluate three specific optimizations made possible by this new form of compiler/processor collaboration.

- **Physical Register File Reduction.** Superscalar processors remove false dependences by mapping architectural register names to a large file of physical registers. Often on a processor's timing critical path, the physical register file must be managed conservatively to maintain precise program state. We demonstrate that the use of DVI allows the processor to reclaim physical registers aggressively, allowing the register file to be smaller and faster, potentially increasing processor clock rate.
- **Dead Save/Restore Elimination in Procedure Calls.** The use of procedures facilitates programming but incurs substantial save and restore overhead. Traditional static analysis that attempts to minimize this overhead based on static liveness information is inherently restricted in scope and precision. We present a simple hardware technique that uses *dynamic* liveness information to minimize this overhead at runtime.
- **Dead Save/Restore Elimination across Context Switches.** Multi-threaded programs

incur save and restore overhead on thread switches. Preemptive switches, in particular, are not amenable to static optimization. We show that our hardware scheme is easily extended to minimize thread switch overhead dynamically.

DVI can be calculated at compile time. We suggest that current instruction set architectures (ISAs) should be extended to enable the communication of DVI to the processor. Using the above techniques, DVI can improve system performance with minimal added hardware complexity. In addition, we describe a DVI implementation that supports our optimizations, while incurring minimal runtime overhead.

The rest of this paper is organized as follows. Section 2 describes sources of DVI, compiler support, and encoding issues. Our experimental framework is described in section 3. The next three sections motivate, describe, and evaluate each of our proposed optimizations in detail. Section 4 describes register file size optimizations. Section 5 deals with save/restore elimination for procedure calls. Save/restore elimination in thread-switches is covered in section 6. We use section 7 to address a variety of issues, most notably the overhead of explicit DVI instructions. Section 8 presents related work. We discuss implications of our work and conclude in section 9.

## 2 DVI Basics

DVI is available to the processor both explicitly and implicitly. We now describe both forms.

**Explicit DVI.** In general, DVI must be explicitly encoded into the executable. Encoding *explicit DVI* (E-DVI) is done using *E-DVI instructions* which are added to the ISA. An E-DVI instruction explicitly states that a register is dead at that point in the program. Figure 1 shows an E-DVI instruction, E3, marking the death of the callee saved register r16 after its last use. Our implementation of E-DVI instructions defines a subset of the non-opcode bits as a *kill mask* for a register subset, with a register dead if the corresponding bit is set.

The information encoded in E-DVI instructions is computed using static, intra-procedural liveness analysis performed in standard compilers [7]. However, a full compiler is not necessary to encode E-DVI. Since liveness information is computed for physical registers, E-DVI instructions can be added to an executable using a simple binary rewriting tool. This approach is attractive since it requires neither compiler nor program source code.

Runtime overhead added by E-DVI instructions is an important implementation consideration. Fortunately, since E-DVI is not required for correct execution, a range of E-DVI is possible, from none to frequent E-DVI instructions. E-DVI should be inserted into a program binary only to the extent that its overhead can be overcome by the optimizations it enables. Our implementation inserts a single E-DVI instruction which contains a kill-mask for the callee-saved registers before every procedure call. We found this strategy to be effective for our optimizations and inexpensive. We discuss E-DVI overhead in greater detail in section 7.

| | Dynamic Instruction Stream | | Register(s) |
|---|---|---|---|
| | `main:` | `proc:` | Killed |
| I1: | r16 ← | | |
| I2: | ← r8, r16 | | |
| **E3:** | **kill r16** | | r16 (E-DVI) |
| **I3:** | **call** `proc` | | r8 (I-DVI) |
| I4: | | r8 ← | |
| I5: | | ← r8 | |
| **I6:** | | **return** | r8 (I-DVI) |
| I7: | r16 ← | | |

Figure 1: **E-DVI and I-DVI example.** In this example r8 is caller-saved while r16 is callee-saved. r16 is killed *explicitly* by the E-DVI instruction E3. r8 is killed *implicitly* by I-DVI deduced from the call (I3) and return (I6).

**Implicit DVI.** E-DVI can provide arbitrarily detailed DVI, but incurs a runtime overhead. However, using dynamic execution cues and the machine language/ABI calling convention, the processor can infer a DVI subset at no overhead. Standard RISC calling-conventions define a set of *caller-saved* registers whose values are dead at the entry and exit points of any procedure. A dynamic instance of a call or return instruction provides *implicit DVI* (I-DVI) for these registers. In figure 1 the procedure call to `proc` (and corresponding return) kills the caller-saved register r8. I-DVI incurs no runtime overhead, and requires no changes to the executable or the ISA. However, I-DVI provides information only for the caller-saved registers. Because I-DVI is available only at procedure calls and returns, it is most useful when procedure call frequency is high.

## 3 Experimental Framework

In this section we describe our simulation environment, our benchmark suite, and our conventions in reporting experimental results.

**Simulation Environment.** To evaluate our optimizations we used the SimpleScalar tool set [2]. The detailed out-of-order processor simulator was modified to support MIPS R10000-style register renaming [10]

| Parameter | Value |
|---|---|
| Issue Width | 4 |
| Inst. Window | 64 |
| Func. Units | 4 int (2 mul/div), 2 fp (1 mul/div) |
| Cache Ports | 2 (fully independent) |
| L1 D-Cache | 64KB, 4-way, 1 cycle latency |
| L1 I-Cache | 64KB, 4-way, 1 cycle latency |
| L2 Cache | 512KB, 4-way, 8 cycle latency |
| Branch Predictor | 16-bit history, BTB, 256K entry combinational gshare/bimod |

Figure 2: **Machine configuration.** Machine parameters used in our simulations. The values were chosen to be representative of current high-performance uniprocessors such as the MIPS R10000 [10] and DEC Alpha 21264 [11].

| Benchmark | Dynamic Inst | Call Inst | Mem Inst | Saves & Restores |
|---|---|---|---|---|
| compress | $0.5 \times 10^9$ | 0.7% | 9.5% | 0.0% |
| go | $0.4 \times 10^9$ | 1.1% | 28.8% | 4.0% |
| ijpeg | $0.6 \times 10^9$ | 0.6% | 27.1% | 4.5% |
| perl | $1.0 \times 10^9$ | 1.3% | 47.1% | 9.2% |
| gcc | $1.0 \times 10^9$ | 1.2% | 40.1% | 12.7% |
| vortex | $1.0 \times 10^9$ | 0.8% | 53.1% | 17.7% |
| li | $1.0 \times 10^9$ | 1.7% | 45.6% | 11.1% |

Figure 3: **Benchmark characterization.** Dynamic instruction count, and calls, memory references, and saves and restores as a percentage of total dynamic instructions.

and to exploit DVI. Figure 2 presents our specific machine parameters.

**Benchmark Programs.** We used seven integer benchmarks from the SPEC95 benchmark suite: `compress95`, `go`, `ijpeg`, `li`, `vortex`, `perl`, and `gcc`. The benchmarks were compiled using a modified version of GNU GCC-2.6.3 at the `-O2` optimization level. The standard libraries were not recompiled to include DVI, possibly limiting our results. All benchmarks were simulated to completion or up to 1 billion instructions (100 million for the register file optimizations results) and used the reference data inputs, except for `go` (`30 10 null.in`) and `li` (`test.lsp`). Figure 3 provides a brief characterization of the benchmarks.

**Significance of Results.** For all of our evaluations, the IPC figures we report are *original* program instructions per cycle, a true measure of the work done by the program. We do not count E-DVI annotations as instructions executed, considering them as cycle overhead only. Our baseline simulations always use binaries which do not contain E-DVI annotations.

| | Dynamic Instruction Stream | Physical Mapping for *r1* | Free Register List |
|---|---|---|---|
| I1: | r1 ← | p1 | p2 p3 ... |
| I2: | ← r1 | p1 | p2 p3 ... |
| I3: | kill r1 (implicit or explicit) | (none) | p2 p3 ... p1 |
| | [Instructions that do not use or set r1. These may contain branches and loops] | | |
| I4: | r1 ← | p2 | p3 ... p1 |

Figure 4: **Register File Optimization Example.** An ordinary processor can only reclaim the physical register p1 (allocated for r1) when I4 commits. Using DVI allows the processor to reclaim p1 when I3 commits, allowing p1 to be used in renaming the intermediate instructions.

## 4 Register File Size Reduction

Access to a large, multi-ported physical register file is an important element of the timing-critical path in a multiple issue processor [6, 3, 8]. Access time is quadratic in the number of read and write ports and linear in the number of registers [6]. Current register file optimizations reduce the number of ports through replication or pipelining. Our technique reduces the number of registers in the file and can be used in conjunction with other cycle time optimizations.

The physical register file must be large enough to hold *all* architectural values and *all* renamed destinations in the instruction window. The use of DVI allows the processor to free storage containing *dead* architectural values more quickly, allowing the use of a smaller register file without restricting renaming or limiting instructions per cycle (IPC).

The dynamic code sequence in figure 4 demonstrates the operation of our scheme. When I1 is renamed, a physical register (say p1) is allocated to hold the value of r1. Even though the value in p1 is not needed after I2, an ordinary processor cannot reclaim p1 until an instruction assigning to the same architectural register commits, such as I4. As a result, p1 cannot be used in renaming the instructions between I3 and I4. The number of dynamic instructions between I3 and I4 can be arbitrarily large. Using DVI the processor can free p1 when I3 commits. Between I3 and I4 the architectural register r1 is not mapped to any physical register.

### 4.1 Hardware Support

To support early reclamation of physical registers, standard register renaming hardware can be extended to exploit DVI. We add a single state bit to each entry of the architectural-to-physical register mapping table. The bit is set when the value in the register is live and clear otherwise. Collectively, we refer to this set of bits as the *Live Value Mask* (LVM). The LVM is updated at the decode stage by destination renaming and instructions that provide DVI, explicitly or implicitly. Since the freeing of a "dead" physical register is an unrecoverable action, physical registers can only be reclaimed when the corresponding DVI instruction is known to be non-speculative.

### 4.2 Evaluation

Current processors are designed with sufficient registers (64-80 physical registers) such that program IPCs are not constrained by register renaming resources. Therefore our DVI enabled register file optimization will not increase peak IPC. However, reducing the size of the physical register file may enable an increased clock rate, possibly providing performance gains. To isolate the effect of this optimization and present our results in a more meaningful context we assume that the physical register file cycle time is proportional to the processor's overall cycle time. We evaluate the effectiveness of our optimization both in terms of register file size reduction and in terms of overall system performance (IPC $\times$ clock rate) improvement.

Since our optimization reduces the size of the physical register file and all benchmarks must use the same size file, we must define a meaningful domain for measuring performance. For this reason, we compute all relevant quantities over an "average workload," which we define as the unweighted arithmetic mean over all benchmarks simulated to 100 million instructions.

**Reduction of Register File Size.** To quantify the file size reduction directly, we measure IPC for a range of physical register file sizes with no DVI, I-DVI only and both I-DVI and E-DVI. As figure 5 shows, the use of I-DVI allows our benchmarks to achieve roughly 90% of peak IPC at register file sizes only a little larger than the minimum of 32 required to avoid deadlock. The E-DVI instructions we insert before procedure calls have little added value, which leads us to believe that a high density of E-DVI is necessary to provide any appreciable additional benefit.

**System Performance Improvements.** To compute overall performance (IPC $\times$ clock rate), we use a modified version of CACTI[13, 5] to generate a timing model for multiported register files (a 4 way issue machine requires 8 read ports and 4 write ports) of different sizes. System performance is then computed by dividing the IPC curves of figure 5 by the access time computed by the register file timing model. The resulting curves in figure 6 show system performance as a function of register file size. The overall system performance improvement made possible by our optimization can be calculated as the ratio between the peak performance values for the optimized and unoptimized configurations. These peaks occurs at the optimal number of registers for a design based on our assumptions. When our DVI-enabled technique is used, the size of the physical register file at peak performance decreases from 64 to 50 registers, a 22% reduction, and overall performance improves by 1.1%.

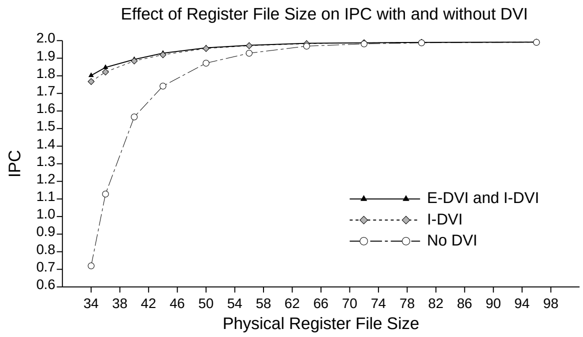


Figure 5: **Average IPC as a function of register file size.** Shows the IPC averaged over all benchmarks as a function of the size of the integer physical register file.

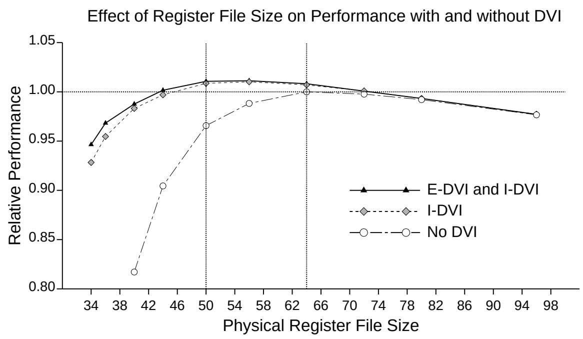


Figure 6: **Performance as a function of register file size.** Shows the overall system performance calculated by dividing the IPC by the register file cycle time. The data is scaled relative to the peak performance with no DVI (the horizontal line). The two vertical lines mark the peak performance with DVI (50 registers) and without DVI (64 registers).

## 5 Dead Save/Restore Elimination in Procedure Calls

The only registers that need to be preserved across a procedure call are those that are both (i) live at the *caller* and (ii) used in the *callee*. To use these heuristics intra-procedurally, ISAs define a *calling convention* that divides the general purpose registers into *caller-saved* and *callee-saved* sets. Caller-saved registers obey only criterion (i); they must be preserved by the caller if their value is live across a call. Compilers greedily use these registers to hold temporaries and values that are not live across calls. Callee-saved registers consider only criterion (ii) and must be saved by the callee if they are used by it. Compilers prefer to hold values that are live across calls in these registers.

Figure 7(a,b) shows two calls to the same procedure `proc`. Register r16 is live in `caller1`, requiring the save (I3) and restore (I6) in `proc` to be executed. However, r16 is dead in `caller2`, and the same save and restore are now unnecessary. Since `proc` does not "know" who is calling it, it must *always* save and restore r16 so that it executes correctly if called from `caller1`. Calling conventions and their associated heuristics are unable to handle *context-sensitive liveness*. Nor can traditional interprocedural analyses [14, 17] cope with this problem. This difficulty can only be overcome statically by compiling multiple caller-specific procedure versions (*clones*), each containing the appropriate save/restore sequence. However, cloning expands code and can have adverse effects on I-cache performance.

We propose a simple hardware/software technique that overcomes these limitations without the need for interprocedural analysis or cloning. We use DVI to track register value liveness along a *dynamic path* and eliminate save/restore pairs of callee-saved registers at runtime.[1] Our solution, shown in figure 7(c), involves inserting an E-DVI annotation which kills r16 before the call to `proc` in `caller2`. The LVM hardware structure keeps track of the fact that the *value* in r16 is dead and dynamically eliminates the save (I3) and restore (I6) of that *value*. Since r16 is live at `caller1`, a corresponding DVI instruction is not inserted, and I3 and I6 execute normally when `proc` is called along that path.

While static techniques remove unnecessary saves and restores from execution completely, our solution still requires these instructions to be fetched and decoded. However, since dead saves and restores are not dispatched, our method frees up bandwidth to the L1 cache, and effectively increases commit bandwidth and the size of the instruction window.

### 5.1 Software Support

Our scheme requires two software components to enable save/restore elimination. First, saves and restores must be implemented using new store and load variants. *Live-loads* and *live-stores* only execute if their data registers are marked live. In figure 7 the save at I3 and the restore at I6 would be encoded as a live-store and live-load. For flexibility we introduce new instructions rather than add semantics to the callee-saved registers. Adding the new instructions allows the compiler to specify uniform behavior for all registers, and different behavior for the same register at different program points.

Save/restore elimination targets the callee-saved registers. Unfortunately, I-DVI provides information only about caller-saved registers. The second requirement, therefore, is that the compiler insert E-DVI for the callee-saved registers into the executable. Two observations allow us to minimize the amount of E-DVI that must be inserted. First, save/restore elimination requires information only at call sites, bounding the amount of overhead to a single E-DVI annotation per dynamic instruction call. Further, E-DVI must be inserted only if a callee-saved register is both assigned to in the procedure and dead at the call site.

### 5.2 Hardware Support

We present two hardware schemes for save/restore elimination. The first scheme builds on the LVM structure of section 4.1 and is used to eliminate saves of callee-saved registers. The second scheme adds a stack to buffer LVM information from procedure entry points and can be used to eliminate both saves and restores.

**LVM Scheme.** Elimination of saves can be performed using the LVM structure introduced in section 4.1. Added decode logic detects live-store instructions whose data register is marked dead in the LVM and does not dispatch them. Figure 8 shows the running code example in (a) and the operation of the LVM scheme in (b). The kill mask in I3 sets the LVM bit to D(ead), allowing the save instruction within the procedure to be eliminated.

**LVM-Stack Scheme.** We wish to eliminate a restore whenever its matching save has been eliminated. For an implementation to do this, it must eliminate restores based on the same LVM bits used to eliminate the corresponding saves at the procedure entry. The LVM itself is updated continuously as a procedure executes and cannot be used directly for this purpose. As shown in figure 8(b), the LVM loses track of the live bit used to eliminate the save (I3) and thus cannot be used to eliminate the matching restore (I6).

An LVM-Stack is used to overcome this limitation. The LVM-Stack buffers an LVM "snapshot" from the procedure entry until its exit. Restores are eliminated based on the information at the top of the LVM-Stack, as this is the same information used to eliminate the matching saves. The operation of the LVM-Stack in

[1] A related optimization can be used to eliminate restores of caller-saved registers, but since it does not use DVI we do not discuss it here.

| | caller1: | proc: |
|---|---|---|
| I1: | ← r16 | |
| | | *(r16 live)* |
| I2: | call `proc` | |
| I3: | | save r16 |
| I4: | | r16 ← |
| I5: | | ← r16 |
| I6: | | restore r16 |
| I7: | | return |
| I8: | ← r16 | |

(a)

| | caller2: | proc: |
|---|---|---|
| I1: | ← r16 | |
| | | *(r16 dead)* |
| I2: | call `proc` | |
| **I3:** | | **save r16** |
| I4: | | r16 ← |
| I5: | | ← r16 |
| **I6:** | | **restore r16** |
| I7: | | return |
| I8: | r16 ← | |

(b)

| | caller2: | proc: |
|---|---|---|
| I1: | ← r16 | |
| **E2:** | **kill r16** | |
| I2: | call `proc` | |
| **I3:** | | ~~**save r16**~~ |
| I4: | | r16 ← |
| I5: | | ← r16 |
| **I6:** | | ~~**restore r16**~~ |
| I7: | | return |
| I8: | r16 ← | |

(c)

Figure 7: **Save/Restore Elimination Example.** (a) shows a call to `proc` from `caller1` where r16 is live. (b) shows a call to `proc` from `caller2` where r16 is dead, using a single conservatively compiled version of `proc`. In this case, I3 and I6 (in bold) are executed needlessly. (c) shows `caller2` again, this time with a kill instruction inserted before the call allowing I3 and I6 to be eliminated. The saves and restores are implemented using live-store and live-load instructions, respectively.

| | caller2: | proc: |
|---|---|---|
| I1: | ← r16 | |
| **E2:** | **kill r16** | |
| I2: | call `proc` | |
| **I3:** | | **save r16** |
| I4: | | r16 ← |
| I5: | | ← r16 |
| **I6:** | | **restore r16** |
| I7: | | return |
| I8: | r16 ← | |

(a)

| LVM |
|---|
| L |
| L |
| D |
| **D** |
| D |
| L |
| **L** |
| L |
| L |

(b)

| LVM | LVM-Stack | |
|---|---|---|
| L | . . | *stack grows* → |
| L | . . | |
| D | . . | (1) *push* |
| D | . . **D** | |
| D | . . D | |
| L | . . D | (2) *maintain* |
| L | . . **D** | (3) *eliminate* |
| L | . . D | (4) *pop* |
| D | . . | |

(c)

Figure 8: **LVM and LVM-Stack schemes working example.** (a) shows a dynamic code sequence with a dead save/restore pair. (b) shows the state (Live/Dead) of the r16 bit of the LVM *before* each instruction in the LVM scheme. (c) shows the same bit for the LVM and the LVM-Stack in the LVM-Stack scheme.

conjunction with the LVM is shown in figure 8(c). At a procedure call, the current LVM is pushed onto the LVM-Stack (1). An assignment to register r16 sets the live bit in the LVM, but the same bit at the top of the LVM-Stack is unchanged (2). The restore can now be eliminated using the LVM-Stack bit (3). Finally, at the return, the LVM-Stack is popped and its contents copied back into the LVM (4).

Implementations of hardware stack mechanisms are well-understood. We simulate a small circular buffer which wraps around on overflow and assumes an empty stack on underflow. Our simulations use a 16-entry LVM-Stack. Our studies show that a 16-entry mechanism captures nearly 100% of the benefit of an unbounded size structure on all benchmarks except for `li` where 94% of the benefit is achieved.

### 5.3 Evaluation

We begin our evaluation of save/restore elimination by directly measuring its effectiveness in eliminating save and restore instructions. Next, we evaluate its impact on IPC. Finally, we perform a sensitivity analysis to measure its interaction with relevant microarchitectural parameters.

**Dynamic Saves and Restores Eliminated.** The fraction of saves and restores eliminated is a property of the program and the amount of available DVI. It is independent of the processor configuration. Figure 9 shows dynamic saves and restores eliminated as a percentage of total dynamic callee saves and restores, total memory references, and total instructions. We present our results for the six benchmarks that exhibit significant save and restore activity. The LVM-Stack scheme, which handles both saves and restores, eliminated 46.5% of all dynamic save and restore instructions, 11.1% of all memory references, and 4.8% of all instructions, respectively. The numbers are most striking for `perl`, in which 74.6% of callee saves and restores and 7.2% of total instructions need not be exe-

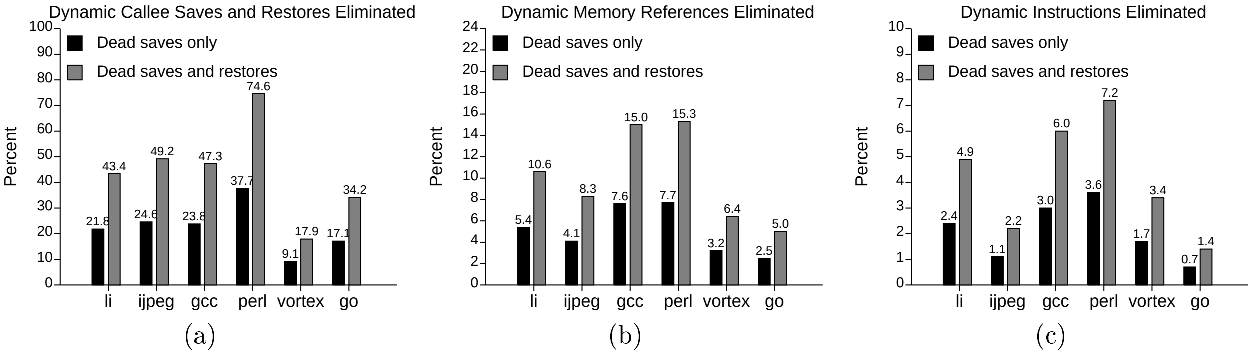


Figure 9: **Dynamic Saves and Restores Eliminated.** Shown as a percentage of (a) total saves and restores, (b) total memory references and (c) total instructions. We show the instructions eliminated using both the LVM scheme, which eliminates saves only, and the LVM-Stack scheme which eliminates both saves and restores.

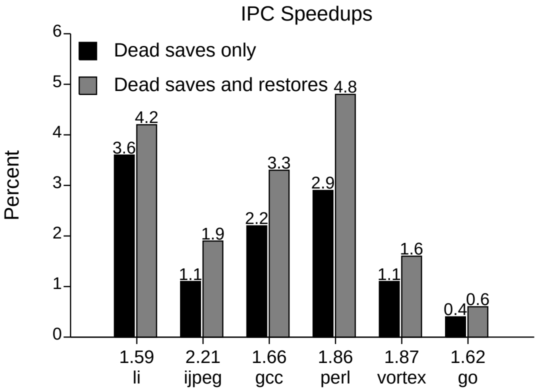


Figure 10: **IPC Speedups.** Shows the IPC speedups for both the LVM and LVM-Stack schemes. The base IPC is given at the base of each pair of bars.

cuted. The LVM scheme, which eliminates only saves, provides half the benefit. These numbers illustrate the inefficiencies associated with calling conventions and intra-procedural static techniques.

**IPC Speedups.** We now examine how effective removal of dead saves and restores translates into IPC improvements. Figure 10 shows for each benchmark the IPC gains achieved by eliminating saves only using the LVM scheme, and by eliminating both saves and restores using the LVM-Stack scheme.

We expect IPC gains to be proportional to the percentage of total instructions eliminated for each benchmark, but diminished by the fact that the total number of instructions fetched and decoded is not reduced. Our results support this intuition. `gcc`, `perl`, and `li` see the greatest reduction in instructions executed and the greatest increase in IPC, with `perl` leading the way at 4.8%. In addition, we expect a high degree of correlation between IPC speedups and the percentage of total memory references eliminated, since one important effect of save/restore elimination is reducing cache bandwidth requirements. Again, the same three benchmarks support our intuition, with `perl` translating a 15.3% reduction in memory references into a 4.8% IPC increase. The fact that save elimination accounts for more than half of the IPC benefit is due to the diminishing ability to exploit the extra bandwidth afforded by restore elimination.

**Sensitivity Analysis.** By definition, the memory references removed by save/restore elimination have no true data dependences. The primary benefit of this optimization, therefore, is in reducing data bandwidth requirements. Consequently, we expect the IPC gains to be sensitive to the relationship between requirements of the program and the data bandwidth supplied by the processor.

Increasing the number of cache ports is an effective way of providing high data bandwidth. Multiple cache ports can be implemented through *replication* or *banking*. The former is free of bank contention and provides more bandwidth while the latter requires significantly fewer implementation resources. For most programs a banked implementation can realistically achieve the performance of 3 perfect cache ports [18]. As our simulations model a replicated (perfect) cache, we expect our performance improvements to be more significant on a more realistic, lower bandwidth banked configuration. While adding cache ports increases the available bandwidth, increasing the issue width increases cache bandwidth requirements. Figure 11 shows the performance of two benchmarks for different cache port/issue width configurations. As expected, the relative effectiveness of save/restore elimination increases

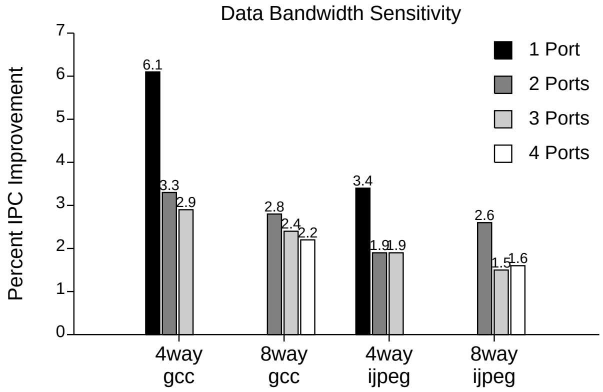


Figure 11: **Cache Bandwidth Sensitivity Analysis.** Sensitivity of save/restore elimination optimization to number of cache ports and issue width.

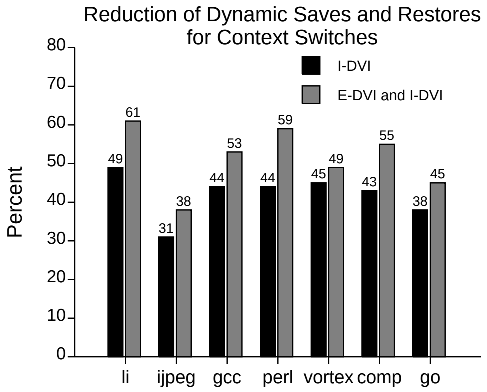


Figure 12: **Context Switch Saves and Restores Eliminated.** Shows the percentage of saves and restores that can be eliminated dynamically at context switch time (per benchmark).

as the number of cache ports decreases. Increasing the issue width produces two opposing effects: an increase in commit bandwidth reduces the effectiveness of save/restore elimination, while an increase in data bandwidth required makes it more effective. In general, for configurations at which a program is bound by data cache bandwidth, this optimization provides a significant benefit.

Our studies show that the effectiveness of dead save/restore elimination is insensitive to other related microarchitectural parameters including the size of instruction window and reorder buffer.

# 6 Dead Save/Restore Elimination Across Context Switches

Process and thread switches normally require that the architectural processor state, including the values of all architectural registers, be preserved. While the cost of saving and restoring this state is not significant for context switches [15], it dominates thread switch overhead especially for fine-grained threaded code [1, 9]. Non-preemptive switches are implemented using a procedure call interface allowing the compiler to generate specialized save and restore code at these well-defined switch points based on static liveness information [9]. Preemptive switches are not amenable to such static analysis or optimization and must conservatively save and restore all registers.

We propose that DVI be used in multi-threaded programs to optimize saves and restores dynamically. Unlike previously proposed solutions, our solution does not require whole program analysis or procedure cloning and handles preemptive switches.

## 6.1 Software and Hardware Support

The LVM scheme presented in section 5.2 can be used to support save elimination across context switches. The only requirement is that the software routine performing the thread switch must implement saves and restores using live-load and live-store instructions described in section 5.1.

As we saw earlier, eliminating a dead save requires only dynamic information about the liveness of the data value being saved. Eliminating a dead restore, however, requires that we locate the matching save and consult its liveness status. The same holds for restore elimination across thread switches. Since procedure calls and returns follow a stack pattern and are hardware-managed, a hardware stack can be used to eliminate restores. Thread blocks and resumes are arbitrarily ordered and managed by software, requiring a general-purpose software mechanism to implement restore elimination. The basis of a simple solution requires a pair of instructions which would allow the LVM to be saved to and loaded from the thread or process control block. An *LVM-save* instruction would be issued before a context switch, and an *LVM-load* instruction should be issued before all register restores when returning to the context.

## 6.2 Evaluation

We report the performance improvement achieved in terms of the percentage reduction in the average number of integer register saves and restores executed at context switches. The number of registers saved and restored is computed by generating a histogram of the number of live architectural registers and calculating the average number of registers holding live values during execution.

| Bench | Dyn. Inst. Count | Static Code Size | Instructions per Cycle 4-way set associative 32K I-cache | 64K I-cache |
|---|---|---|---|---|
| li | 1.7% | 1.0% | 1.26% | 1.22% |
| ijpeg | 0.6% | 1.2% | 0.00% | 0.01% |
| gcc | 1.2% | 4.0% | -0.02% | -0.33% |
| perl | 1.3% | 2.6% | -0.12% | -0.01% |
| vortex | 0.8% | 2.9% | -1.63% | -0.40% |
| go | 0.7% | 1.9% | -0.25% | -0.10% |

Figure 13: **E-DVI Overhead.** Shows the percentage overhead (per benchmark) in dynamic instructions fetched and static code size, and the corresponding overheads in IPC for two cache configurations. A negative overhead signifies an IPC increase.

The results are summarized in figure 12. For these benchmarks, the average number of integer saves and restores executed across context switches decreased by 51% when E-DVI instructions were inserted before procedure calls. Using I-DVI only, we were able to achieve a 42% reduction. This is a significant savings and should easily offset the added overhead of saving and restoring the LVM. In addition, floating point registers are often dead in integer codes and thus most of the saves and restores of floating point registers can be eliminated.

## 7 Implementation Issues

We now address several DVI implementation concerns mentioned in previous sections.

**E-DVI overhead.** E-DVI overhead can be separated into two effects. Primarily, E-DVI increases the number of dynamic instructions that must be fetched and decoded. The second effect arises due to an increase in static code size, decreasing the effective capacity of the instruction cache. We quantify these effects and their impact on IPC. To do so, we compare the IPC values for executables with and without E-DVI in the absence of our DVI-enabled optimizations. Intuitively, IPC overhead should be proportional to and *smaller than* the percentage overhead in dynamic instruction count, since E-DVI annotations are effectively no-ops.

As figure 13 shows, E-DVI overhead (or impact) is negligible and is due primarily to an increase in dynamic instruction count. Small increases in code size perturb I-cache alignment and instruction fetch, producing only slight fluctuations in performance. This result underscores the fact that E-DVI overhead is not a fundamental problem. E-DVI instructions do not introduce false dependences or consume functional or renaming resources. While they do increase the number of instructions that must be fetched, they do not increase the number of branches. A sufficiently large I-cache and fetch queue can absorb most of this cost.

**Meaning of precise program state.** Our register file optimizations rely on creating situations where certain architectural register names are not bound to values. What is the meaning of precise program state in this scenario? By definition, the meaning of any dead value is irrelevant to the remaining execution of the program. In that sense, *any* value assigned to an unbound architectural name results in correct execution.

**Hardware and ABI interactions.** In order to deduce I-DVI from call and return instructions the processor must know which registers are caller-saved, a set defined by the ABI calling convention. To avoid ABI dependence, I-DVI should be inferred for those registers set in an ABI supplied mask. A clear mask indicates that no I-DVI should be inferred, and can be used in debugging.

**Effect of DVI on program correctness.** Although DVI is not required for the correct execution of the program, it has a definite (and often unrecoverable) effect on processor state. Incorrect E-DVI will almost certainly lead to incorrect execution; the compiler is held responsible to provide only correct E-DVI. Errors in E-DVI should be considered compiler errors.

**Speculative updates of hardware structures.** LVM and LVM-Stack updates occur at decode time and are often *speculative.* To ensure correct execution in the event of mis-speculation, these structures can be checkpointed and recovered by the same mechanism which supports such actions for the mapping table. This same mechanism can keep track of reclaimed physical registers, conserving space in the reorder buffer.

**Non-standard call-return sequences and context switches.** Exceptions, non-standard call-return sequences (i.e. `longjmp()`), and even context-switches disrupt the function of the LVM and LVM-Stack mechanisms. A simple strategy to handle this class of events would be to flush these structures and safely assume that all registers are live. Alternatively, support could be added to save and restore the contents of the LVM (as is already required for our restore elimination across thread switches) and the LVM-Stack.

## 8 Related Work

The idea of using DVI for register file optimizations is not new. Sohi and Franklin [8] study register instance lifetimes and describe how compiler support can be used to minimize these lifetimes and reduce traffic. Lozano and Gao [3] use DVI to reduce write-back traffic between the reorder buffer and the physical register file. More recently, the Multiscalar [16] architecture uses summary masks to streamline register communication. These techniques try to reduce the number of datapaths to the register file. Our method reduces the size of the physical register file.

The VAX [4] uses compiler-supplied masks to encode saves and restores in the callee based on intra-procedural *use* information. Huguet and Lang [12] extend this mechanism to eliminate some of these encoded saves and restores in hardware dynamically. In their solution, called Policy-G, register values are saved when they are overwritten by the callee and restored on a demand basis in the caller. This strategy is based on dynamic register use rather than on compiler communicated, hardware-tracked liveness. Policy-G can effectively eliminate restores in a flow-sensitive manner which our technique cannot handle, but does not eliminate saves based on liveness. In addition, Policy-G requires a large amount of critical path hardware, and more involved changes to existing instruction sets.

Kurlander and Fischer [14] use interprocedural analysis and profile information to produce a statically optimal interprocedural spilling strategy. Their technique attacks all registers, but does not consider path information and must therefore produce conservative save/restore code. Our solution deals with callee-saved registers only, but dynamically streamlines save/restore code using runtime information without the need for interprocedural analysis. Kurlander and Fischer report an average 5% reduction in execution time on an inorder machine. Our 4.8% average reduction in dynamic instruction count is a comparable result.

Grunwald and Neves [9] use interprocedural analysis to determine the live registers at each non-preemptive thread-switch call site and compile custom save-restore code for each call. Their solution requires cloning and does not handle preemptive switches. Our method, on the other hand, does not require cloning or interprocedural analysis and easily handles preemptive switches.

## 9 Conclusions

We make the following contributions in this paper:

- We describe the concept of Dead Value Information (DVI). We introduce minimal ISA extensions that can encode DVI efficiently. We also observe that implicit DVI is inherent in programs due to the ISA calling conventions.
- We describe a technique that uses DVI to reduce the size requirements of the physical register file by efficiently reclaiming physical registers which contain dead values. We show that decreasing register file size potentially increases system clock speed, and improves overall performance.
- We show that a simple hardware mechanism can be used to track DVI and eliminate saves and restores to callee-saved registers dynamically. We demonstrate that our software/hardware technique handles situations not handled by software alone, and dynamically eliminates 46% of static save/restore code. On some benchmarks, our method achieves IPC improvements of nearly 5%.
- Using the same hardware scheme, we show that save/restore elimination can be extended to handle both non-preemptive and preemptive context switches. Our results show an average reduction of 51% in the number of integer saves and restores.
- Moreover, we demonstrate that our optimizations rely on well-known compiler techniques and only minor ISA and hardware modifications.

Our work on save/restore elimination concentrates on performance improvements in the context of current calling conventions and the standard strategies for register allocation, scheduling, etc. Our proposed ISA extensions and hardware mechanisms give selected registers the desirable property of only being saved when required based on dynamic path information. The implications for register allocation, the use of calling-conventions, and future ISA design need to be explored.

Our current implementation places E-DVI instructions before procedure calls. While this encoding effectively supports save/restore elimination across procedure calls, it is most likely insufficient for other optimizations, especially in programs with few procedures. Further study is required to assess the benefit versus cost of other encoding strategies. Interesting design points include placing E-DVI instructions at the beginning and/or end of loop bodies or entire loops.

Object oriented languages such as C++ and Java are gaining wide acceptance. These languages contain features such as dynamic binding and linking which make whole-program analysis and optimization nearly impossible, and a rise in procedure call frequency inevitable. Save/restore elimination will become even more effective in these execution models. Java is especially amenable to DVI-based optimization due to its support of threads.

## Acknowledgments

This work was supported in part by NSF Grants CCR-9505922, CCR-9509589, and MIP-9505853, by the U. S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0217 and DARPA order D346, and by donations from Intel. Milo Martin is supported by an IBM Graduate Fellowship. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

The authors would like to thank Jim Goodman, Guri Sohi, Doug Burger, Stefanos Kaxiras, Subbarao Palacharla, Manoj Plakal, Andreas Moshovos, Scott Breach, and E. Ender Bilir for their comments on early revisions of this paper, and the anonymous referees for their suggestions. We thank Norm Jouppi and Keith Farkas for assistance with the register file timing model.

## References

[1] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System*, volume 26, pages 108–121, April 1991.

[2] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, Madison, WI, June 1997.

[3] Luis A. Lozano C. and Guang R. Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 292–302, November 29–December 1, 1995.

[4] Digital Equipment Corporation. *VAX-11 Architecture Reference Manual*, 1982.

[5] Keith I. Farkas. *Memory-System Design Considerations for Dynamically-Scheduled Microprocessors.* PhD thesis, University of Toronto, 1997.

[6] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register file design considerations in dynamically scheduled processors. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, January 1996.

[7] Charles N. Fischer and Richard J. LeBlanc Jr. *Crafting a Compiler with C.* Benjamin/Cummings Publishing Co., 1991.

[8] Manoj Franklin and Gurindar S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, December 1–4, 1992.

[9] Dirk Grunwald and Richard Neves. Whole-program optimization for time and space efficient threads. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 50–59, 1–5 October 1996.

[10] Linley Gwennap. Mips R10000 uses decoupled architecture. *MicroProcessor Report*, pages 18–22, October 24 1994.

[11] Linley Gwennap. Digital 21264 sets new standard. *MicroProcessor Report*, pages 11–16, October 28 1996.

[12] M. Huguet and T. Lang. Architectural support for reduced register saving/restoring in single-window register files. *ACM Transactions on Computer Systems*, 9(1):66–97, February 1991.

[13] Norman P. Jouppi and Steven J.E. Wilton. An enhanced access and cycle time model for on-chip caches. Technical Report 93.5, DEC Western Research Laboratory, July 1994.

[14] Steven M. Kurlander and Charles N. Fischer. Minimum cost interprocedural register allocation. In *The 23rd Symposium on Principles of Programming Languages*, January 1996.

[15] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, April 1991.

[16] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar architectures. In *The 22nd International Symposium on Computer Architecture*, 1995.

[17] David W. Wall. Global register allocation at link time. Research Report 17, Digital Western Research Laboratory, September 1989.

[18] Kenneth Wilson and Kunle Olukotun. Designing high-bandwidth on-chip caches. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 2-4 1997.