Discovering Barriers to Efficient Execution, Both Obvious and Subtle, Using Instruction-Level Visualization

David M. Koppelman
Division of Electrical & Computer Engineering
Louisiana State University, Baton Rouge, LA 70803 USA

Chris J. Michael
Naval Research Laboratory
Stennis Space Center, MS 39529 USA

***Abstract*—CPU performance is determined by the interaction between available resources, microarchitectural features, the execution of instructions, and by the data. These elements can interact in complex ways, making it difficult for those seeing only aggregate performance numbers, such as miss ratios and issue rates, to determine whether there are reasonable avenues for performance improvement. A technique called instruction-level visualization helps users connect these disparate elements by showing the timing of the execution of individual program instructions. The PSE visualization program enhances instruction-level visualization by showing which instructions contribute to execution inefficiency in a way that makes it easy to locate dependent instructions and the history of events affecting the instruction. A simple annotation system makes it easy for a user to attach custom information. PSE has been used for microarchitecture research, simulator debugging, and for instructional use.**

I. Introduction

CPU performance is determined by a number of interacting factors; this is particularly true for dynamically scheduled processors in which instructions can start execution out of order. For those tuning performance or designing the next generation of processors, conventional methods of performance evaluation do not present the information needed by programmers or architects to quickly grasp unexpected problems or delve deeper into more subtle issues.

Popular existing tools, such as HPCToolkit [1], are certainly helpful for finding where possible bottlenecks might be, even down to the source line, or with data-centric tools [2], down to a variable. However, knowing that execution time, cache misses, or branch mispredictions are associated with a particular line of code or variable is not enough to identify many issues.

For example, consider a high-level code expression that uses variables of different sizes. A compiler might emit a long sequence of instructions to ensure that the result is "correct" in the case of overflow, rather than a short sequence which would be correct for the program. The user would be unlikely to find such an error using existing tools, let alone determine whether these instructions were on a critical path and so costing one or more cycles per instruction, versus a typical cost of a quarter cycle for non-critical instructions.

This paper introduces PSE, an instruction-level visualization system which enables the user to not just find areas of inefficient execution, but to help in finding the underlying cause. PSE, though in use for over ten years, is not the first instruction-level visualization system. Stolte *et al* in 1999 [3] and Weaver *et al* [4] in 2001 describe earlier systems, and the popular SimpleScalar simulator collection has long had a visualization tool, ss-vis [5].

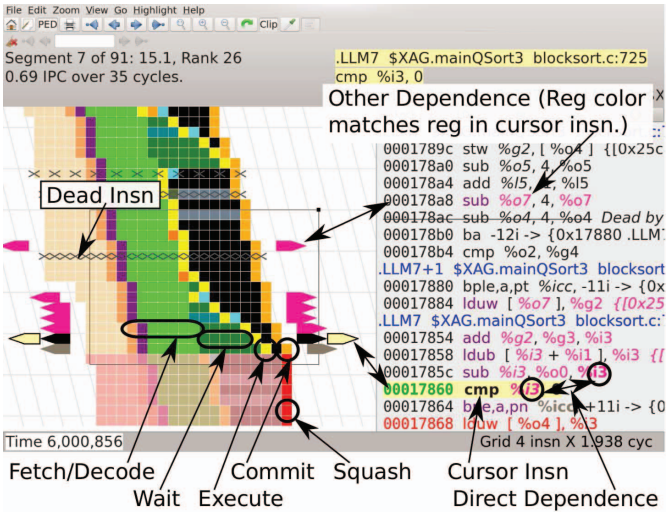

Fig. 1. PSE displaying a PED (pipeline execution diagram) plot from bzip2 program with the PED view zoomed to show detail.

In instruction-level performance visualization the events occurring in the execution of individual dynamic instructions are plotted over time. See Figure 1. Events correspond to pipeline stages, load instruction progress (L1 lookup, L2 lookup, etc), non-pipeline states (*e.g.,* awaiting operand, awaiting commit), etc. In PSE these are displayed in a way that highlights the interaction between instructions and brings out the inefficiencies in execution. The instructions are plotted alongside the disassembled code, and this disassembled code includes procedure names and source line numbers (if available). The instruction-level display is complemented by an overview display that shows performance over the entire execution of a program.

It is this display of assembly code and the use of dynamic instruction dependence highlighting that would make problems such as the unnecessarily long sequence of integer instructions

978-1-4799-7058-2/14 $31.00 © 2014 IEEE
DOI 10.1109/VPA.2014.11

easy to spot. Other problems stand out too; several examples are discussed in Section V.

PSE has multiple features to help the user delve deeper into understanding code execution, including multiple display modes and animation, as well as an annotation mechanism facilitating the collection of data for the investigation of ad-hoc hypotheses. These enable, for example, the user to view the recent past of an instruction under scrutiny (cache hits for load instructions, mispredictions for branch instructions, etc.).

The goal for code tuning is both to quickly discover the obvious such as the forgotten use of a *restrict* qualifier and the subtle, such as predication's benefit. PSE helps with both. For computer architecture research it provides the complementary benefit of showing inefficiencies that can be fixed not by code tuning but by new instructions or microarchitectural features.

PSE has been used for some time for many architecture investigations including SW v. HW prefetch, out-of-order fetch mechanisms, and branch predictions. It has also been used for teaching classes in computer architecture at several levels.

The data for a PSE visualization is collected from a CPU simulator. Currently PSE is being used with RSIML, a simulation of a dynamically scheduled SPARC implementation. Simulation, rather than gathering data from a CPU, is unavoidable because of the high volume of data that needs to be recorded. (Impressive things can be done with event register sampling [6], but these techniques still cannot capture an actual sequence of events that might be important to characterize a program.)

II. Background

A. Dynamically Scheduled Processors

A *statically scheduled* CPU core starts execution of instructions in program order, whereas *dynamically scheduled* (also called out-of-order) cores start the execution instructions when their operands are ready, often avoiding *stalls* that would occur in statically scheduled cores. Dynamically scheduled cores make sense for code with less predictable branch and cache behavior, and are the dominant organization for general-purpose CPUs. Both organizations can be n*-way superscalar*, meaning that they can sustain execution at a rate of up to n instructions per cycle on certain code sequences under favorable conditions.

For a number of reasons, the out-of-order scheduling in dynamically scheduled cores makes it difficult for many programmers to connect program characteristics to their performance impact, something which PSE helps with.

Here we will briefly describe execution in such a processor. Those less familiar with dynamic scheduling might consult some early descriptions of dynamically scheduled processors [7], [8], more recent designs [9], and a study of the inherent differences between static and dynamic designs [10]. The *front end* of the core is responsible for fetching, decoding, and enqueuing an instruction for execution, a process which takes several cycles and which here will be referred to as *fetch/decode*; see Figure 1. When an instruction's source operands are available and a functional unit is available the instruction can start *execution*, which can take one or more cycles. The last step in an instruction's lifetime is called *commit*, which must be performed in program order. At any one time there can be many instructions *in flight* (over 100 on some processors). If there is a problem with one in-flight instruction, such as a misprediction or exception, the execution of following instructions must be *squashed* (erased, or undone). In flight instructions are kept in a *reorder buffer (ROB)*, and consume other resources. When the ROB is full, fetch stalls. Instruction progress through a pipelined core is often hand-analyzed using a *pipeline execution diagram*, in which each instruction occupies a horizontal line and short abbreviations indicate instructions' location in the pipeline. Such diagrams can be extended to track instructions in dynamically scheduled systems. PSE uses the term PED to refer to such a diagram.

The latency of a chain of dependent instructions (or just one instruction) can affect the efficiency of dynamically scheduled systems in two ways: it can cause the the ROB to fill, stalling fetch, or it can end at a frequently mispredicted branch [11]. In both cases reducing the latency (even at the cost of additional instructions) will improve performance.

B. Instruction Set for Code Examples

The examples in the following sections show execution of code on an implementation of the SPARC v8b architecture, a RISC instruction set [12]. For the most part the code should be readable by someone familiar with some assembly language. SPARC v8b has 32 general purpose 64-bit registers named `g0` to `g7`, `o0` to `o7`, `l0` to `l7`, and `i0` to `i7` (for general, output, local, input). (A register windowing scheme allows these to be mapped to different sets of 24 registers, facilitating procedure calls and returns.) There are 32 64-bit floating point registers, `f0`, `f2`, to `f62`. In SPARC assembly language the destination register is last.

C. Hardware Configuration for Code Examples

The execution examples below are taken from the simulation of a 4-way superscalar SPARC implementation, based loosely on a Fujitsu SPARC VI processor [13]. The processor has an 80-slot ROB and uses gshare branch prediction. It can issue up to 4 integer and 2 floating-point instructions per cycle. The L1 data cache is 128 kiB and the L2 cache is 4 MiB, the line size in both caches are 64 bytes.

III. PSE Components and Operation

PSE consists of the *PSE library* for use in CPU simulators, and the *PSE visualization program*. Calls to the PSE library are used to collect events needed for instruction visualization, other time-sampled performance data, and any other information that one may want preserved such as the time the simulation was run. All of this information is written to a *dataset file*, which the PSE visualization program reads.

A. The PSE Library and Collected Data

The highest-resolution data that PSE collects is *event* data. Each event item has a timestamp and can be associated with an instruction or a group of instructions. They are used to identify points in the execution of an instruction, such as entering the decode stage, and to associate information with the instruction

or group of instructions, such as the instruction address. There are pre-defined events that PSE recognizes, such as events for *commit* and *squash*. Others can be user defined. PSE uses events to control plotting in the *PED* and *ROB* views, usually to switch the displayed color.

General information about a dynamic instruction can be attached using an *annotation*. An annotation can be used to place a marker in the plot area, a message in the message area, or to mark up the assembly code for the instruction. The markup can change the color and font style of the assembler code, and can add text before or after the disassembled instruction. Uses include displaying branch outcome history or prefetch effectiveness. It is easy to define new annotations and a UI is provided select which ones are displayed.

This high-resolution data is collected during periods of time called *segments*. Experience has shown that 2000-cycle segments work well. This data undergoes two types of compression, counter-type values such as timestamps and instruction serial numbers are difference coded, then the resulting data is compressed using the bzip2 library. Even so, the amount of data is too large for segments to cover all of execution, even though simulations themselves sample a program's execution. In the examples given here segments are aligned with the samples used by the simulator.

The PSE library also has calls to collect performance data covering each segment. The intent is to characterize the performance over the segment as an aid to the user in locating segments of interest.

PSE also collects non-temporal data, which can be organized hierarchically. When used with RSIML in the simulations reported here and elsewhere PSE collects all simulator output, even stdout, facilitating the archiving of simulation results.

B. The PSE Visualization Program

The PSE visualization program is written in C++ and uses the GTK windowing library. It builds on Intel 64 (x84_64) versions of Red Hat Enterprise Linux 6 and Fedora 20. The name PSE is a pseudo abbreviation of Processor Simulation Elucidator, an earlier name was SEE which was changed to PSE about the time people started using search engines as their primary means of finding things. Development versions can be obtained using Subversion from the link https://svn.ece.lsu.edu/svn/dmk/trunk/pse. A large collection of datasets can be found under "Simulation Data Repositories" at http://svn.ece.lsu.edu/. The benchmarks for these are at /svn/dmk/trunk/benchmarks/simtools/.Benchmark_Archive in the same repository.

The PSE visualization program reads the dataset file and also tries to locate the executable file of the simulated program which is needed for disassembly and providing source information. PSE can setup and maintain a benchmark archive for the executables, in which they will be named with an MD5 hash of their contents. For a description of functionality see Section IV.

IV. Using PSE

The PSE visualization program is used to browse the results of a simulation (possibly in progress) recorded in a dataset

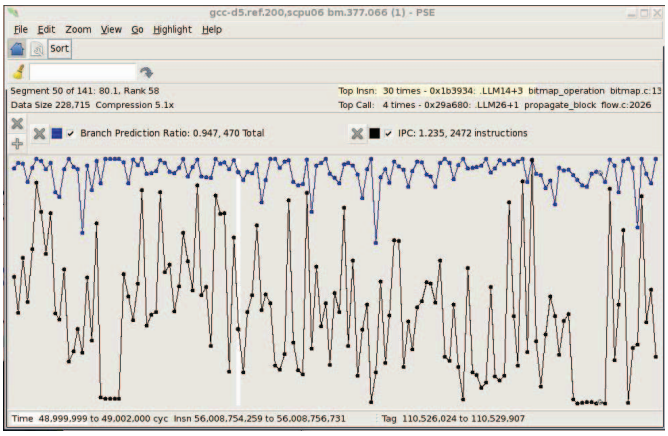

Fig. 2. PSE displaying an overview plot for the gcc benchmark. The black series is execution rate in instructions per cycle (maximum value is 4), the blue series shows the branch prediction ratio.

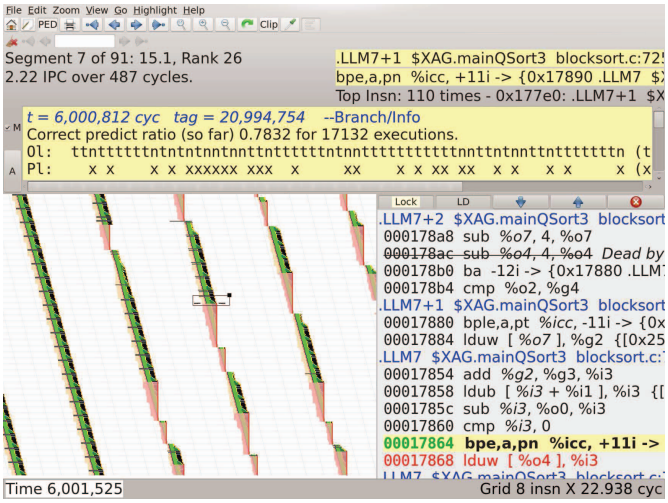

Fig. 3. PSE displaying PED plot for bzip2 benchmark. The x axis is time, each instruction occupies a different y position with wraparound. The message window (yellow background) in this example shows outcome and prediction history for a branch (the one with the yellow background in the disassembly window and at the bottom of the small boxed area in the PED plot).

file. When the file is first loaded the user is presented with an *overview plot* which shows values of various performance measures over time, see Figure 2. All are plotted on the same axis, which by convention is scaled so that the maximum y value indicates 100% efficiency.

The user can choose which series to show. Initially one might choose execution rate (IPC) along with those series with a major impact on performance such as branch prediction ratio and cache hit ratios. The plot is initially shown in time order, the plot can also be sorted by execution rate, enabling one to visualize how the other series correlate with performance, and by code location, allowing users to compare similar areas. The plot in Figure 2 is sorted by time.

The overview plot helps the user characterize execution over time and to quickly determine where the program is inefficient or exhibiting other behaviors of interest.

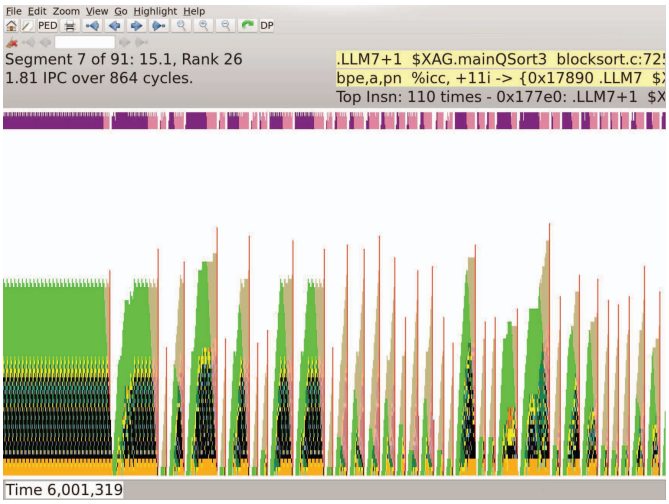

Fig. 4. PSE displaying ROB plot for bzip2 benchmark. The purple in the band along the top indicates an instruction entering decode. Light purple indicates decode of an instruction that will later be squashed (discarded). Gaps are due to a number of factors such as misprediction recovery and instruction alignment. If execution were occurring at the peak rate the band would be solid purple. Colors below the purple band show the status of instructions that have passed through decode.

Clicking on one of the points displayed in the overview plot brings the user to the *segment* window, see Figure 3. The segment window has three major panes, the *instruction plot pane* (lower left) initially shows timelines in the execution of instructions over the period covered by the pane, the *disassembly pane* (lower right) shows disassembled code corresponding to the displayed instructions (or instructions nearby), and the *message pane* (center, with yellow background) shows textual information items (called *annotations*) corresponding to an instruction, time, or other entity.

The instruction plot pane shows what will be called a *PED view* (for pipeline execution diagram). It can be switched to a *ROB view* in which an instruction's vertical position is based on its position in the reorder buffer (a FIFO structure), see Figure 4. The ROB view emphasizes the number of instructions in flight, and it is plotted in a way that also emphasizes fetch/decode (front-end) efficiency.

The transition between the ROB view and PED view is animated, helping the user see the connection between the ROB view, which emphasizes resource usage, and the PED view, which emphasizes instruction interaction.

Each dynamic instruction occupies a horizontal line, with horizontal positions corresponding to time. This can be clearly seen in Figure 1 where the PED view has been zoomed and labels have been added. In the PED and ROB views colors indicate events and states. In the figure there are labels for those colors which are needed for understanding the examples provided here. The colors pointed to by *Fetch/Decode* include all of the processing steps needed to prepare for execution, *Wait* indicates an instruction is waiting for a source operand, *Execute* indicates that an instruction is executing (load instructions use additional colors to indicate their progress), *Commit* indicates that an instruction will no longer be needed, and *Squash* indicates that an instruction was fetched by mistake and its effects must be undone. The colors for an instruction are tinted red if that instruction will be squashed, as are the instructions at the bottom of the figure.

Instructions in the plot pane can be highlighted by *markers*, and instructions in the disassembly pane can be highlighted in a number of ways (such as by background color).

A user-movable *instruction cursor* points to a dynamic instruction, it is labeled *Cursor Insn* in Figure 1. The corresponding instruction is highlighted in the plot pane (with a yellow marker) and disassembly pane (with a yellow background).

Markers and highlights are applied to instructions that the cursor instruction depends upon (pink in the figure), and instructions that depend on the cursor instruction (brown in the figure). This includes dependencies through memory. Other dataflow related information is shown. For example, a register is shown in italic if that is the last use of its value, and dead instructions (those that write values that are never used) are shown crossed out.

Additional markers and disassembly highlighting can be added by user-defined *annotations*: PSE API calls which attach information to an instruction or time. Annotations can also place a message in the message pane. Annotations are used for showing information related to branch prediction and for both hardware and software prefetch, for example. In Figure 3 an annotation is used to show branch behavior. Annotations were designed to be easy to use so as to encourage ad-hoc investigation.

V. Examples

To show what can be done with instruction-level visualization the execution of three programs will be analyzed, bzip2, gcc, and milc, all as packaged with the SPECcpu2006 benchmarks. The execution will be on a four-way superscalar dynamically scheduled processor implementing the SPARC v8b instruction set.

The programs will be analyzed from the point of view of code tuning and of processor design. In both cases we will be searching for areas of execution that are underutilizing resources. For code tuning the term resources refers to the hardware as it is, for processor design it refers to the hardware that could be designed.

Like many programs, each suffers moderately from branch misprediction and cache misses, and so executes well below the four instruction per cycle peak of the machine. Numbers such as the branch prediction ratio or the L1 cache miss rate are of little help in finding sources of inefficiency. The best modern tools will either attribute such numbers to high-level code items, such as source lines or variables [1], or plot timelines [14]. This will show the user where inefficiencies are occurring but will not provide enough information to clearly identify the issue.

In the examples below execution inefficiencies are discovered using PSE. The user starts in the overview plot, looking at performance measures sampled over program execution, sorted by time, value, or program location. The user then chooses a segment with disappointing performance. The segment plot should quickly tell the user the basic cause: branch misprediction, cache misses, or instruction latency. But by examining

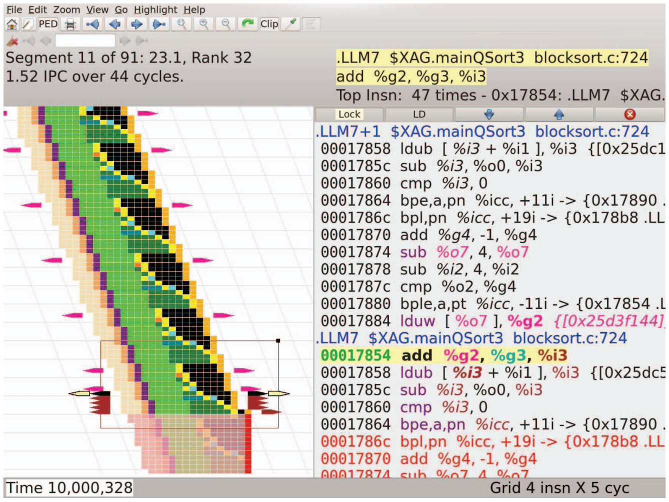

Fig. 5. A code sequence that ends with the misprediction of a late-resolving branch. Examination of the code reveals opportunities to reduce the number of instructions leading to the branch.

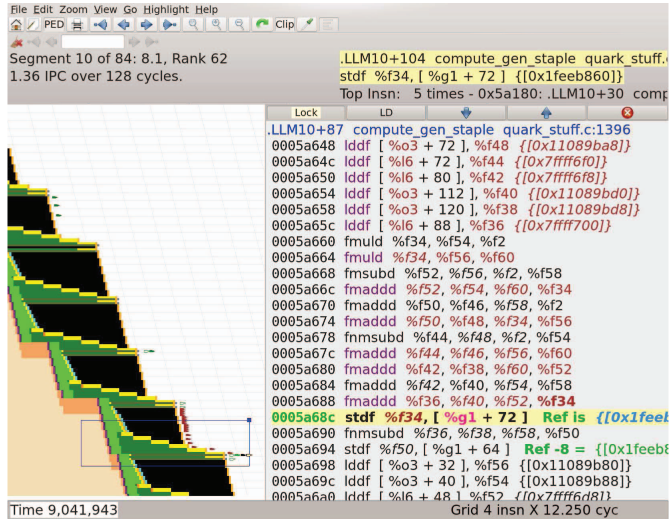

Fig. 6. Code suffering from full ROB stalls. These could be avoided by better scheduling, which the compiler may have avoided due to caution over aliasing.

instruction execution details the user can determine whether these are avoidable.

The issues in the examples that follow could certainly have been discovered by other means, but with PSE these problems can be discovered without specifically looking for them.

A. BZIP2 Branch Latency

Figure 5 shows a PED view for an area of bzip2, the same code covered in Figure 1. Towards the bottom execution ends in a squash. One thing that stands out here is that the mispredicted branch is resolving late, in fact, later than it has to. The yellow blocks denoting execution suggest a dependence chain leading to the branch, and that can be verified in the disassembly window, where the instruction cursor is in the middle of the chain of instructions producing a result. Those familiar with assembly language can note several ways the dependence chain can be shortened. One way is to precompute the sum `g3` plus `i1`. The dependence highlighting lets us quickly determine that `g3` and `i1` are not computed nearby (no green markers in the visible instructions). The `sub` and `cmp` instructions can be combined, and it may be possible to hoist the `lduw` before the branch and use an offset.

All together these would save three cycles per misprediction. With PSE the late resolution stood out and dependence highlighting made it easy to track the instructions involved. It is much less likely that someone would discover this problem by studying assembly language output of a compiler.

Using PSE annotations one can inspect performance history of the branch predictor. The annotations for this branch, shown below, show the 64 most recent outcomes of this branch on the `Ol` (outcome local) line, and the 64 most recent outcomes of any branch on the `OG` (outcome global) line. An `x` on the line below indicates a misprediction. From the recent history it appears that the predictor (which is gshare) could have done better. If one wanted to investigate this branch further one could search for all dynamic instances of this branch.

```
t = 10,000,256 cyc   tag = 34,858,455    --Branch/Info
Correct predict ratio (so far) 0.7687 for 52290 executions.
Ol: tntnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
Pl:       x x                  xx                                   x
OG:    tnntntnntnnnttntntnntntnntntnnttnnntnntntnttnnntnntnntnntntt
Pg:    xx x    xx    x    x  x  x x x    x      x  x                x
```

B. milc example

A different problem is revealed in the execution of milc in Figure 6. Notice that fetch stalls regularly. This is because the ROB is filling with instructions. A larger ROB would solve the problem but would waste energy. There is a long latency dependence path carried by fused multiply add instructions (the wide yellow lines). The code appears to be the result of unrolling a loop.

Dependence highlighting quickly reveals that the different dependence chains are not dependent on each other. Had the compiler overlapped the loop bodies the latency problems would be avoided. Experienced readers may have identified the prime culprit: the compiler was not sure if the loads and stores overlapped. A solution would be to use a *restrict* qualifier (or to use it properly).

One can easily verify that aliasing (address overlap) does not occur in this portion because load and store addresses are shown by PSE. The green "Ref" text indicates whether an address is on the same cache line as the line accessed by the cursor instruction, that's true for the nearby store but not for the loads. (Obviously, this is no proof that aliasing does not occur.)

VI. Related Work

PSE is not the first, nor is it the only instruction-level visualization tool. Stolte *et al*, developed Rivet, an early visualization system for dynamically scheduled processors [3]. This tool presented visualizations that showed instruction positions in core structures, such as the reorder buffer, over time. It did not show the PED style diagrams used by PSE. Though this might help in understanding how such processors worked it

might make it difficult to grasp what was going on over time since particular instruction events would be visible only for an instant. Rivet does show views more like the PSE ROB plots, but these do little to bring out instruction interactions.

Weaver, *et al* describe *GPV* (graphical pipeline viewer) [4], a visualization tool that shows instruction execution in the same PED style as PSE. A more refined visualization tool, ss-vis, was added to the popular SimpleScalar simulator collection [5]. PSE includes more features to help track interactions between instructions, such as dependence highlighting, and makes it easier to add annotations facilitating ad-hoc exploration.

There has been relatively little work since on instruction-level visualization. Instead, investigators have focused on visualization of parallel systems of one kind or another, and on presenting information collected from event counters [1].

VII. Conclusion

This paper has described PSE, a program for presenting instruction-level visualizations. Such tools are invaluable in helping identify bottlenecks or other inefficiencies that might go unnoticed with coarser views of execution. PSE encourages exploration and presents data in a way which makes it easy to discover a variety of problems.

Acknowledgments

The authors would like to thank other contributors to PSE, including Sheela Doshi, Lauren Hatchel, Kade Soprano, and Nitin Srivastava.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.v22:6

[2] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for numa multicore systems," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342826

[3] C. Stolte, R. Bosche, P. Hanrahan, and M. Rosenblum, "Visualizing application behavior on superscalar processors," in *Information Visualization, 1999. (Info Vis '99) Proceedings. 1999 IEEE Symposium on*, 1999, pp. 10–17, 141.

[4] C. Weaver, K. Barr, E. Marsman, D. Ernst, and T. Austin, "Performance analysis using pipeline visualization," in *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, 2001, pp. 18–21.

[5] D. Burger, T. M. Austin, and S. W. Keckler, "Recent extensions to the simplescalar tool suite," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 4–7, Mar. 2004. [Online]. Available: http://doi.acm.org/10.1145/1054907.1054909

[6] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 357–390, Nov. 1997. [Online]. Available: http://doi.acm.org/10.1145/265924.265925

[7] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, Apr. 1996. [Online]. Available: http://dx.doi.org/10.1109/40.491460

[8] R. E. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, Mar. 1999. [Online]. Available: http://dx.doi.org/10.1109/40.755465

[9] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams, "IBM POWER7 multicore server processor," *IBM J. Res. Dev.*, vol. 55, no. 3, pp. 191–219, May 2011. [Online]. Available: http://dx.doi.org/10.1147/JRD.2011.2127330

[10] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?" in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451143

[11] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: ACM, 2001, pp. 74–85. [Online]. Available: http://doi.acm.org/10.1145/379240.379253

[12] S. Microsystems and F. Limited, *SPARC Joint Programming Specification: Commonality Release 1.04*, 2002.

[13] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, "A 1.3GHz fifth generation SPARC64 microprocessor," in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC '03. New York, NY, USA: ACM, 2003, pp. 702–705. [Online]. Available: http://doi.acm.org/10.1145/775832.776010

[14] NVIDIA Corporation, *Profiler User's Guide V 6.0*, NVIDIA Corporation, 2014.