Design Space Exploration of ILP Engines Optimized for Embarassingly Sequential Codes

Christopher Celio
University of California Berkeley
586F Soda Hall
Berkeley, California 94704
celio@eecs.berkeley.edu

Thibaud Hottelier
University of California Berkeley
595A Soda Hall
Berkeley, California 94704
tbh@eecs.berkeley.edu

ABSTRACT

Cycle-accurate simulation is used to design a high-performance ILP engine that is optimized to run "embarrassingly sequential" codes. This project entails two parts: first, a set of embarrassingly sequential benchmarks is assembled; and second, a variety of pipeline models are simulated. Design points to be explored include out-of-order issue versus in-order issue, data prefetching, branch prediction, and issue width.

Categories and Subject Descriptors

C.1.1 [**Computer Systems Organization**]: Processor Architectures—*Single Data Stream Architectures*; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

microarchitecture, performance

Keywords

pipeline design, embarrassingly sequential, ILP core

1. INTRODUCTION

The maximum theoretical performance of a parallelized application is limited to the performance of its sequential components. Therefore, while future architectures may contain thousands of energy-efficient cores, the performance of future applications may still be limited to the single-threaded performance of the processor.

Many proposed architectures are exploring heterogenous systems in which code is executed on the core best suited to exploit a code's behavior to achieve the best performance per unit energy. For example, vector architectures have been shown to have the best performance and energy efficiencies for code that demonstrates significant data-level parallelism. However, many applications still exhibit at least some portion of "embarrassingly sequential" code that cannot be parallelized or made efficient to execute on a vector core.

This project aims to begin a design space exploration of an ILP core for a future heterogenous, many-core processor that can best exploit the "Amdahl" portions of codes.

1.1 Infinicore Architecture

Current architecture research in the Parlab at UC Berkeley involves the design of the "Infinicore" processor. The ILP core explored in this project will be used as a component of the Parlab Infinicore architecture.

Infinicore is a many-tiled processor. Each tile, shown in Figure 1, contains one ILP core, four vector cores, and a network switch. The main application runs on the ILP core (also known as the "Tile Control Processor"), however, parallelized code can be forwarded to the vector cores for more efficient execution. For example, the operating system, runtime, and just-in-time compiler could execute on the ILP core while the data-parallel segments of the application are pushed to the vector cores.

1.2 Previous Research

This project is partly motivated by previous research that recognized scalar cores may need to be designed differently when used in a heterogenous multicore processor. The study used the Cray Y-MP vector machine and analyzed the instruction mix of the code executed on the scalar control processor[8]. This study suggested that little instruction-level parallelism (ILP) remains when the parallelizable code is extracted and executed in the vector unit. Thus, an ILP core designed for use in a heterogeneous many-core processor and optimized for executing embarrassingly sequential code may look radically different from a general-purpose ILP core that runs a variety of mostly sequentially-expressed software that contains data-level parallelism hidden in the form of ILP.

2. CONTEMPORARY PIPELINE DESIGNS

It is difficult to model pipelines accurately without having a good understanding of the micro-architecture to be simulated. Therefore, we studied a variety of current processors that target different design points to gain a better understanding of the cycle time and functional composition of modern cores.

Section 2.1 covers the ARM processors, which provide some good examples of relatively modern designs that focus on the sub-1W platform. Also, ARM processors typically use an in-order pipeline.

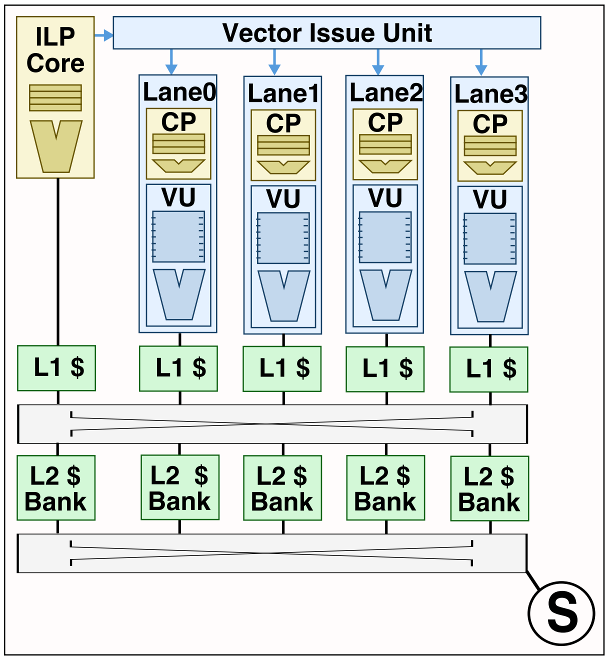

Figure 1: Block diagram of a tile with one ILP core and four vector cores. Each vector core (Lane) consists on an embedded control processor (CP) and a vector unit (VU). All five cores have a private L1 cache, but share the L2 cache. A network switch (S) connects the tile to the rest of the processor.

Section 2.2 covers the POWER5 and POWER6, which aim for the higher-end sever market at around 100 W.

2.1 ARM Series

We examined the micro-architectures of ARM processors from the last 15 years to get an idea of the progression of the designs as technology improved. This allows us to see how heavy constraints in area and power forced different design decisions.

Figure 2 shows the pipeline designs of the ARM7TDMI, ARM9TDMI, ARM11, Cortex-A8, and Cortex-A9 processors. The ARM7TDMI, ARM9TDMI, and ARM11 are in-order and single issue. The Cortex-A8 is dual-issue, but remains in-order, while the Cortex-A9 is dual-issue and out-of-order.

A bigger transistor budget and a drive for a higher clock frequency forced designs with more pipeline stages - from three, to five, to eight, to thirteen - and allows the ARM11 to add multiple functional units that are more specialized for media applications [1, 6], and allows the Cortex-A8 to add a 10-stage SIMD pipeline (not shown in Figure 2).

The Cortex-A8 is a modern low-power, high-performance embedded processor, released in 2008. The core is a super-scalar, 13-stage, in-order design. It has two ALU pipelines and a MAC pipeline, and targets clock frequencies between 600Mhz and 1Ghz[2, 3].

While ARM is reknowned for modern, low-power, in-order designs, it's latest embedded processor uses a speculative, superscalar, out-of-order core: the Cortex-A9. Targeting up to 2 Ghz clock frequencies, the A9 centers around an 8-stage multi-issue pipeline [4].

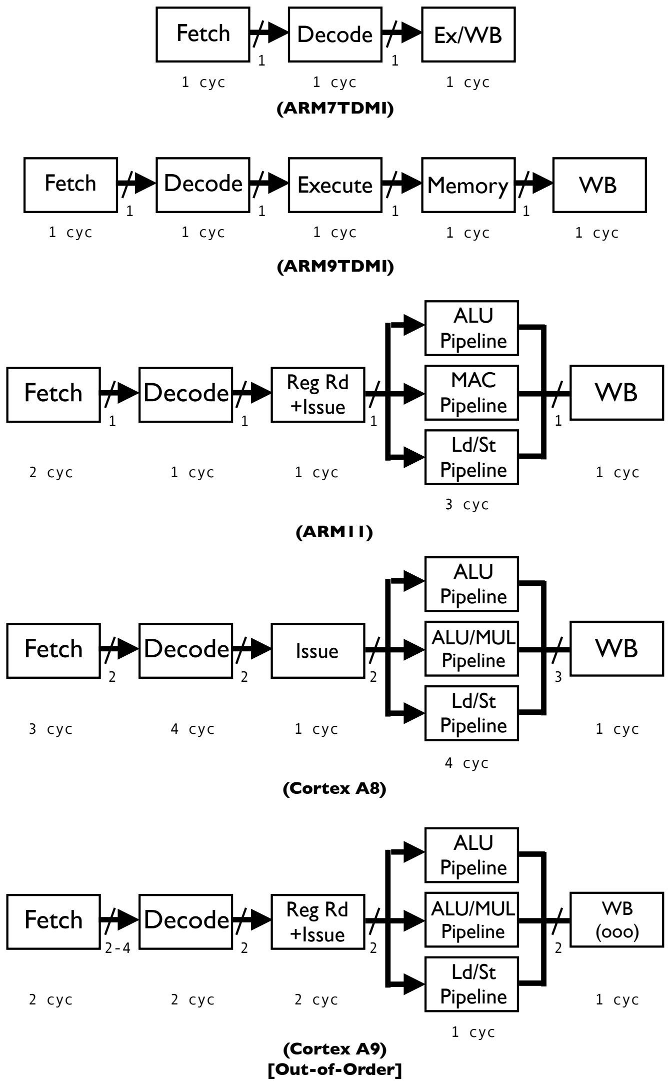

Figure 2: Pipeline diagram of the ARM7TDMI[1], ARM9TDMI[1], ARM11[6], Cortex-A8[2, 3], and Cortex-A9[4] processors. All, except for the A9, are in-order processors.

2.2 Power Series

The POWER5 and POWER6 processors from IBM aim at the 100 W high-end server market. They focus on highly threaded, high throughput environments.

The POWER5 is a simultaneous multithreading, highly speculative, out-of-order processor.

However, the POWER6 explicitly avoids "speculative out-of-order execution that requires costly circuit renaming" in place of a design that "concentrates on providing data prefetch"[5].

Despite this, it does allow the floating point pipeline to go out-of-order to help hide the latency of these longer latency instructions[5].

Because the POWER series is looking at a different application space than our ILP core, we used the ARM processors parameters to guide our pipeline models.

3. METHODOLOGY

This project will use cycle-accurate simulation to explore different pipeline parameters. The SESC simulator, described in Section 3.1, will be used. The design parameters to be explored are covered in Section3.2.

3.1 SESC Simulator

To study the design points discussed in Section 3.2, we chose to use the SESC Simulator[7]. It is a more modern equivalent to SimpleScalar, however, it was designed with an emphasis to allow small scale multi-core simulations to take place. We will not be using that feature for this study, however, SESC allows us to study fully parameterizable in-order and out-of-order superscalar processors. It also includes features such as strided data prefetchers and hybrid branch predictors.

The basic SESC pipeline is shown in Figure 3, along with the parameters chosen for the difference pipeline stage latencies. The instruction latencies are outlined in Tables 1 and 2.

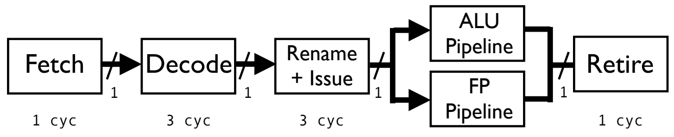

Figure 3: Pipeline diagram of the SESC simulator model used for this study. Functional Unit latencies in the ALU and FP pipelines vary (see Tables 1 and 2).

Instruction	Latency
ALU	1
Store	1
Load	1
Br/Jmp	1
Mult	4
Divide	12

Table 1: Instruction Latencies for the Integer Pipeline.

Instruction	Latency
fpALU	2
fpMult	4
fpDivide	10

Table 2: Instruction Latencies for the Floating Point Pipeline.

3.2 Design Points

The following design parameters are explored using the SESC simulator.

3.2.1 In-Order Versus Out-of-Order

One of the biggest questions in designing a processor involve deciding between an in-order design and an out-of-order design, as it drives further micro-architecture design decisions.

Out-of-order allows the processor to hide the latency involved between dependent instructions by allowing instructions to execute as soon as dependencies have been resolved. The instructions where this matters most are the longer latency instructions such as floating point and load/store instructions. However, "embarrassingly sequential" codes may exhibit less ILP which makes it difficult to schedule longer latency instructions earlier.

As discussed in Section 2.2, some processors can allow a sub-set of instructions to issue out-of-order, while otherwise maintaining an in-order pipeline. Therefore, we also study allowing floating point instructions to issue out-of-order while all other instructions issue in-order. Likewise, we can also allow load/store instructions to issue out-of-order while forcing all other instructions to issue in-order.

Our simulations will use the exact same pipeline for both in-order pipelines and out-of-order pipelines. Only the issue logic will be modified. While choosing how instructions are allowed to be issued will almost certainly change the pipeline micro-architecture, we chose to keep the entire pipeline constant to isolate the effect of out-of-order issue. This is the best that can be done without having more accurate RTL designs to guide the pipeline parameters.

3.2.2 Data Prefetching

Data prefetching is a technique for speculatively prefetching data to hide memory access latencies experienced during cache misses. Because one of the significant benefits for out-of-order cores is hiding memory latencies, data prefetching may be a more energy efficient technique of hiding memory access latencies and thus allow the designer to use a simpler, faster in-order core.

The data prefetcher used for this study will be a strided prefetcher.

3.2.3 Branch Prediction

All but the shortest pipelines are too long for the instruction fetch unit to wait for the resolution of branch conditionals. Therefore, branch prediction is required to keep the pipeline full. However, branch prediction can be expensive and may be unable to predict patterns present in some applications.

For this study we will look at the performance of a hybrid branch predictor (global and local histories) and a static "predict taken" predictor.

3.2.4 Issue Width

Increasing issue width can help keep expensive functional units busy and increase performance by issuing independent instructions simultaneously. However, increasing issue width can be expensive in terms of area and energy. It may also be of little value if applications exhibit very little ILP.

For our simulations, increasing issue width also correspondingly increases the window size and depth, and the number of available functional units. This allows us to better assess the limits of ILP in the applications without being bottlenecked on functional units or other similar parameters.

3.3 Benchmarks

Our benchmarks can be divided in two categories: The first two benchmarks are inherently sequential and hard to express using vector instructions. The other two are program that can take advantage of SIMD instructions. As such we believe that they would be able to take advantage of a vector CPU. All the benchmarks were run on inputs requiring at least 10 million of cycles of computation. The first 100k cycles were thrown away to let the caches warm up.

3.3.1 Picosat

Picosat is a SAT-solver; given a quantifier-free boolean formula, it checks whether the forumula is satisfiable or not. Even though SAT-solvers are relatively small (about 3k SLOC), there are highly-tuned pieces of software. One the key for performance of SAT-solvers is to ensure that all its data structures can fit in the cache(s). This is due to the fact that SAT-solvers have very erratic memory access pattern. As shown in Figure 4 Picosat uses almost exclusively integer operations. The few floating point operations performed collect statistics on the execution and are not in any critical paths of the solver. Picosat is also our benchmark containing the most branches, one every four instructions. In general, SAT-solver are hard to parallelize, because the algorithm and heuristics they use are inherently sequential. We think that it would be very hard to teach Picosat to use vectorial instructions. As such, Picosat would be the kind of workload which would run mainly on the ILP core.

Picosat is run using a formula containing 1132 variables 1264 clauses.

3.3.2 Gzip

Gzip is a well known file compression program. Like most compression program, it is integer heavy. As with Picosat, we think it is not likely that this program can take advantage of vector instructions.

Gzip is used to compress a collection of bash scripts totaling 13.4KB.

3.3.3 Oggenc

Oggenc is a lossy audio file compressor. Unlike the previous two benchmarks Oggenc makes an extensive use of floating point operations. Moreover, even if Oggenc itself does not use SIMD instructions, many other media codecs do. The use of SIMD instructions brings a significant performance gain. Since SIMD instructions can be seen as small vector operations, we are hopeful that Oggenc could take advantage of a vector processor.

Oggenc is run using a very short 3.7KB .wav file.

3.3.4 Blackscholes

Blackscholes is a well known stock pricing simulation. As shown in Figure 4, Blackscholes uses extensively complex floating point operations like division and exponentiation. Blackscholes is also our benchmark with the least number of branches, which means that the control-flow is rather linear and that basic blocks are large. Blackscholes can make use of SIMD instructions, thus we believe that most of the computation performed in Blackscholes could be performed vectorially.

Blackscholes was used to calculate 24 options repeatedly a hundred times.

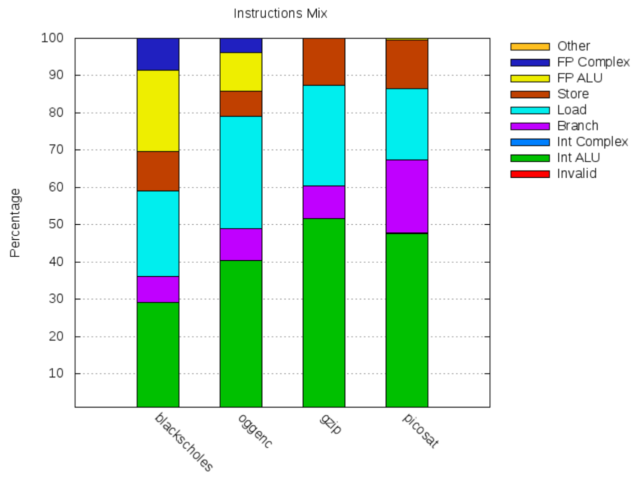

Figure 4: Instruction Mix of the Benchmarks.

4. RESULTS

We compare different pipeline designs by measuring the total cycle time of each run. We further classify how the cycles are spent by measuring the number of retired instructions and by measuring how and why stalls occurred in the pipeline.

"Useful work" is classified as the number of retired instructions divided by issue width. This gives an lower bound measurement of the total number of cycles an application would take if no stalls occurred.

"FU stalls" are measured as stalls that occur in the issue logic. A stall at issue time can be due to something like a busy functional unit, or a dependency hazard that could not be resolved. Most of the stalls for in-order processors are due to "FU stalls".

"Other hazards" are stall cycles inserted due to any other reason, for example, i-cache misses.

The names on the horizontal axis have the following meaning: The first part is the orderness."io" is in-order, "of" is in-order except for floating point operations, "om" is in-order except for loads and stores, and "oo" is fully out-of-order. The second part is the issue width, it ranges from 1 to 4. The third part is prefetching. "nopf" stands for no-prefetching whereas "pf" means that prefetching is enabled. The fourth part has no meaning. Finally the last and fifth part is the branch predictor. "tp" stands for taken predictor whereas "hp" means hybrid predictor.

4.1 Branch Prediction

We studied branch prediction by using the benchmarks Picosat and Blackscholes, as shown in Figures 5 and 6. Picosat and Blackscholes were chosen because they exhibit the extreme between lots of branches (∼25% for Picosat) and relatively few branches (∼5% for Blackscholes), as shown in Figure 4.

The bottom line is that the taken predictor inflicts a dramatic loss in performance across our benchmark suite. So it seems clear that the hybrid predictor is worth the energy and area it uses. Dynamic branch prediction is a requirement for any modern CPUs, even the lowest power ones.

4.2 Data Prefetching

For the applications and data inputs used, prefetching shows very little benefits, as shown in Figures 7, 8 and 9. We expected prefetching to have a significant influence on the performance. We explain our results the following way: Since we have to run the benchmark on relatively small inputs so that they complete is an reasonable amount of time in simulation, we believe that most of the program input as well as its data-structure fits in the cache. That is why prefetching has very little influence in our benchmarks.

4.3 Issue Width

We explore issue width from 1 to 4 for in-order and fully out-of-order processors on all our benchmarks. Results can be seen in Figures 10, 11, 12, and 13. The relative performance gain in out-of-order designs is higher than for in-order designs. This is expected, since out-of-order CPU are more likely to be able to issue as many instruction as possible, by working around program dependencies. The performance gain caused by larger issue width is not great. Hence, the cost-efficiency of wide pipelines is not clear to us.

4.4 In-Order Verses Out-of-Order

The Figures 14, 15, 16, and 17 show the effect of varying "orderness" of the issue logic. First of all, the fully out-of-order pipelines are much faster. The huge performance gain due to going out-of-order can be explained by the high number of integer operation in our benchmarks (at least 30%). We believe that this is not specific to our benchmark selection. Most programs contain a significant number of interger operations. Thus integer performance is critical.

Since out-of-order pipeline are much more complex, our results show that limiting the out-of-orderness to floating point operations or memory operation might be a reasonable trade-off.

5. CONCLUSION

5.1 In-Order Verses Out-of-Order

The results show that in-order cores finish the benchmarks in about 2-3 times more cycles than out-of-order cores. This suggests that out-of-order cores are preferable to in-order cores from a purely performance point of view. However, this assumes that the only change that must occur in the micro-architecture is the issue logic, as was explored in the results shown in Section 4. In reality, in-order cores can typically use shorter pipelines with tighter dependency loops that may gain back some of the performance lost.

Because around half of the instructions issued are integer instructions, there does not appear to be a significant advantage to allowing only some amount of out-of-orderness of either floating point or memory access instructions.

5.2 Data Prefetching

As shown in Section 4.2, strided data prefetching appears to add little to the performance for the pipelines and benchmarks we studied. There are a few reasons that could explain this result.

First, it is possible that we did not simulate enough cycles per run to allow memory prefetching effects to become apparent. Memory access patterns need to be learned before they can be utilized, and if small inputs to the benchmarks can fit entirely within the first-level data cache, prefetching will not be given a chance to prefetch data that may have been previously evicted.

Of course, strided data prefetching does not work if the access pattern is not strided or easily learnable by the hardware. Applications that are more likable to use common and predictable access patterns may be more amenable to data-level parallelism approaches. Typical "embarrassingly sequential applications" may utilize more pointer chasing and irregular memory access patterns that cannot be exploited by strided data prefetchers.

It is interesting to note that the POWER6, ARM11, Cortex-A8, and Cortex-A9 all utilize data prefetching. However, these processors are expected to handle a variety of applications, and not exclusively "embarrassingly sequential" codes.

5.3 Issue Width

Increasing issue width shows only marginal improvements in performance, however, particularly for out-of-order cores, the increase in issue width is expensive in both area and power. Especially if "embarrassingly sequential applications" show little ILP, sticking with a smaller issue width will be a more efficient design point.

5.4 Optimal Design & Future Work

Based on this preliminary study, narrow issue, out-of-order processors with hybrid branch prediction and no data prefetching may be an optimal, energy-efficient design for running Amdahl code.

This study focused on keeping a consistent micro-architecture to study the benefit of out-of-order issue, however, further effort in designing the micro-architecture will be required to clarify the true cost of choosing between in-order and out-of-order. For example, a shorter in-order pipeline may show a more competitive in-order design.

RTL work will begin soon, which will help clarify the design parameters that should be studied in future SESC simulations.

6. REFERENCES

[1] The ARM9 Family - High Performance Microprocessors for Embedded Applications. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 230, Washington, DC, USA, 1998. IEEE Computer Society.

[2] Cortex-A8 Technical Reference Manual Revision:r3p2. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344j/DDI0344J_cortex_a8_r3p2_trm.pdf`, 2010. [Online; accessed 11-May-2010].

[3] ARM. The Cortex-A8 Microprocessor. Technical report, ARM, 2008.

[4] ARM. The ARM Cortex-A9 Processors. Technical report, ARM, 2009.

[5] R. Berridge, R. M. Averill, III, A. E. Barish, M. A. Bowen, P. J. Camporese, J. DiLullo, P. E. Dudley, J. Keinert, D. W. Lewis, R. D. Morel, T. Rosser, N. S. Schwartz, P. Shephard, H. H. Smith, D. Thomas, P. J. Restle, J. R. Ripley, S. L. Runyon, and P. M. Williams. IBM POWER6 microprocessor physical design and design methodology. *IBM J. Res. Dev.*, 51(6):685–714, 2007.

[6] D. Cormie. The ARM11 Microarchitecture. Technical report, ARM, 2002.

[7] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[8] Sriram Vajapeyam and Wei-Chung Hsu. On the instruction-level characteristics of scalar code in highly-vectorized scientific applications. *SIGMICRO Newsl.*, 23(1-2):20–28, 1992.

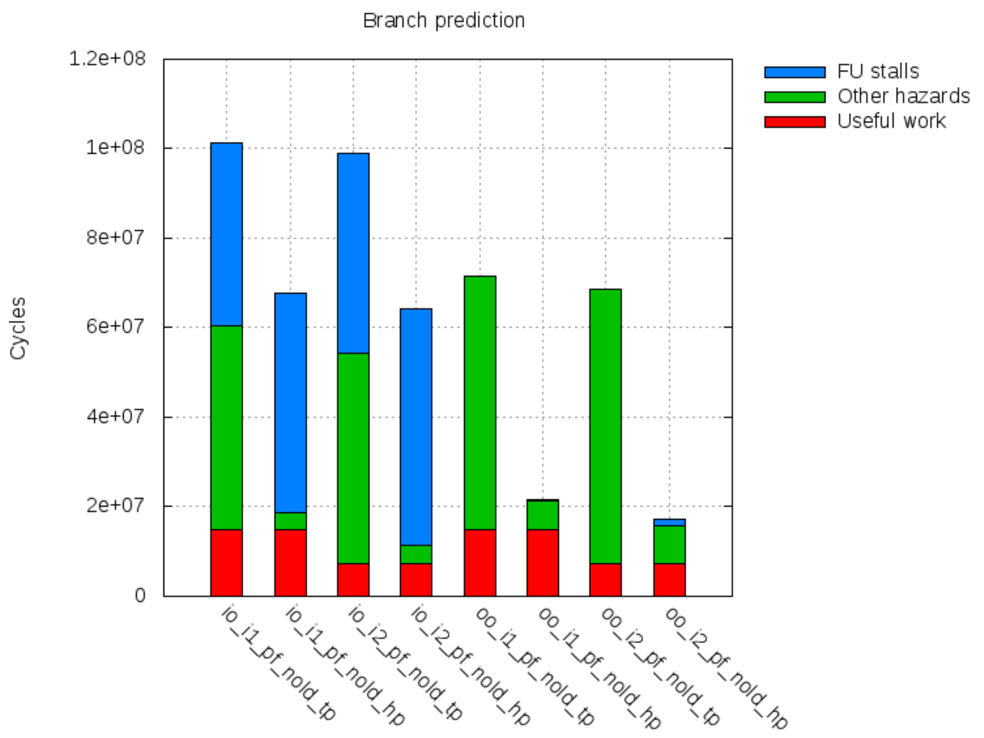

Figure 5: Branch Prediction Results for Picosat.

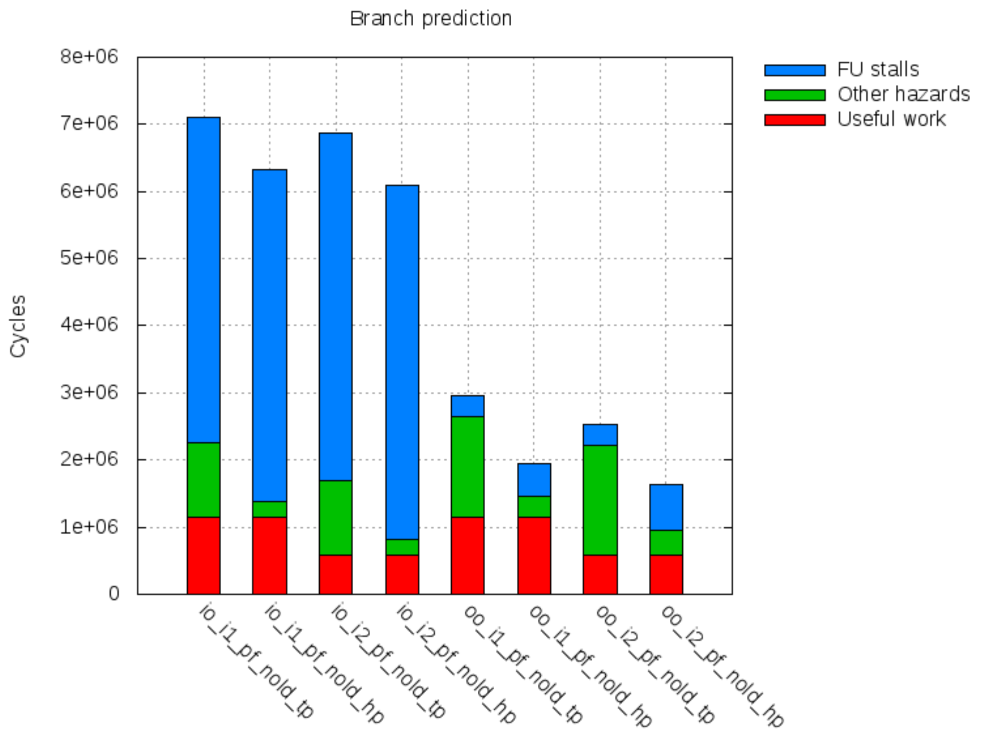

Figure 6: Branch Prediction Results for Blackscholes.

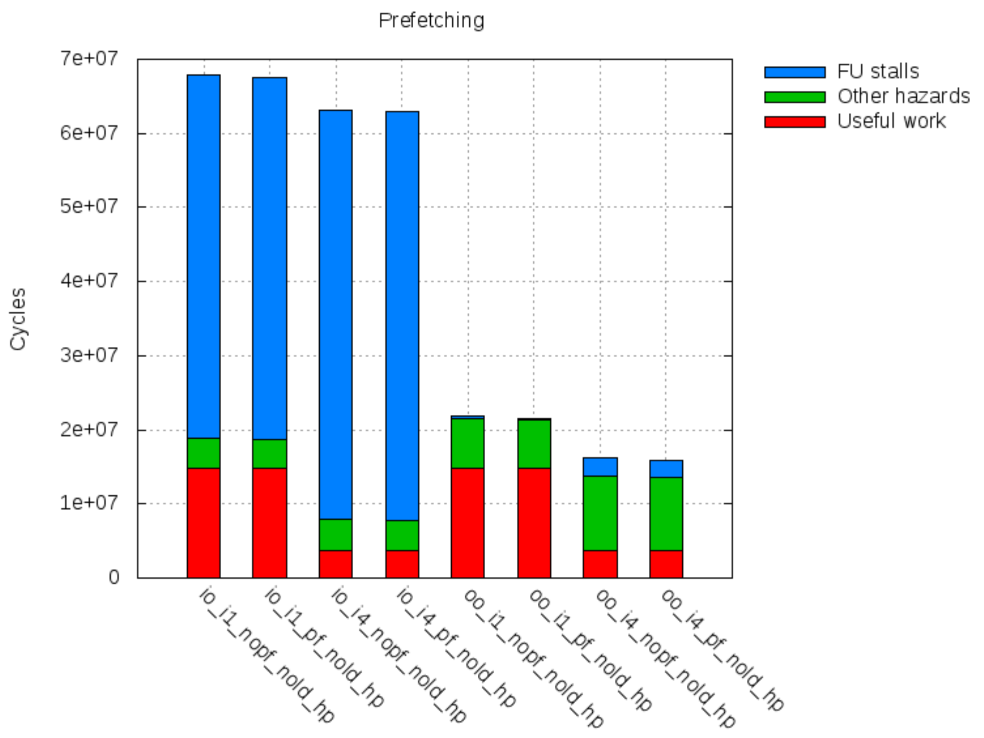

Figure 7: Picosat Results for Data Prefetching.

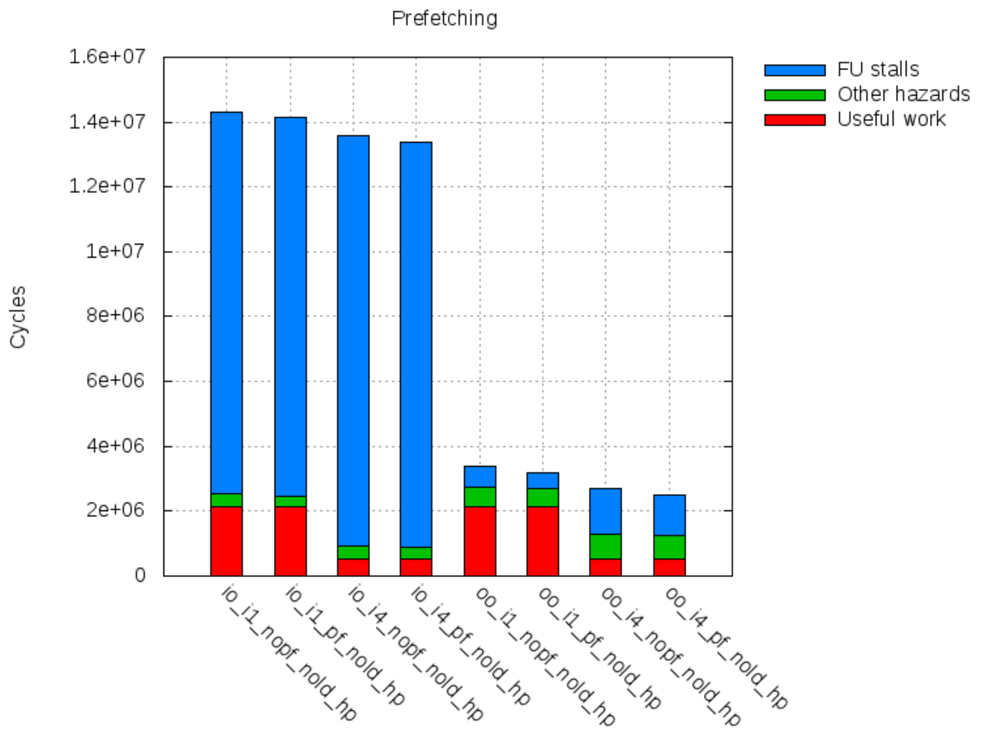

Figure 8: Gzip Results for Data Prefetching.

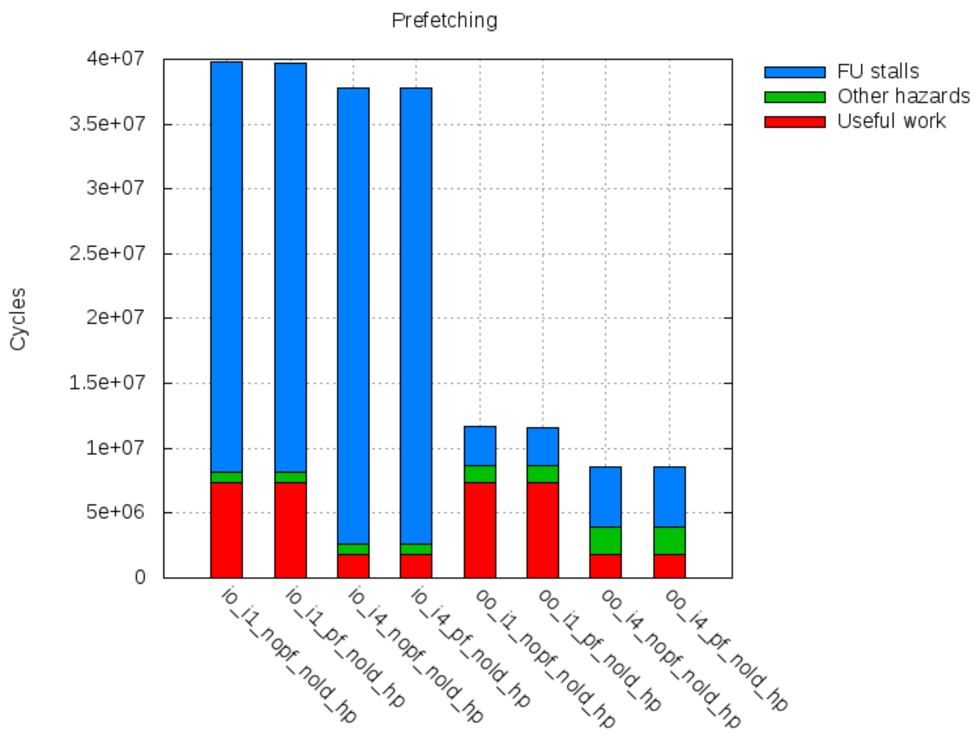

Figure 9: Blackscholes Results for Data Prefetching.

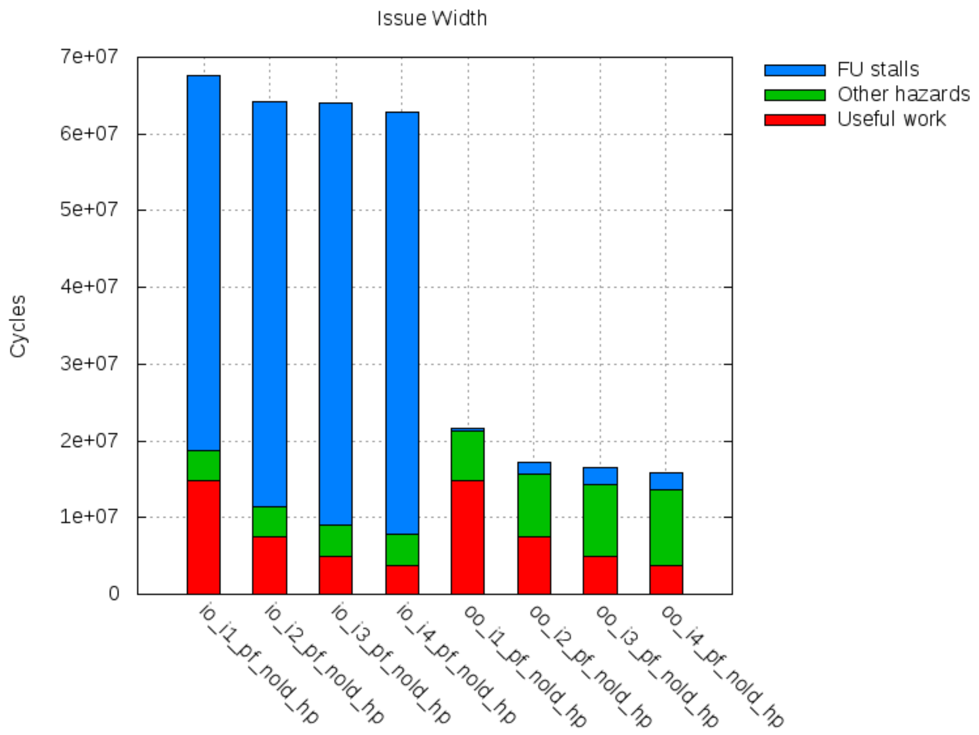

Figure 10: Picosat Results for Issue Width.

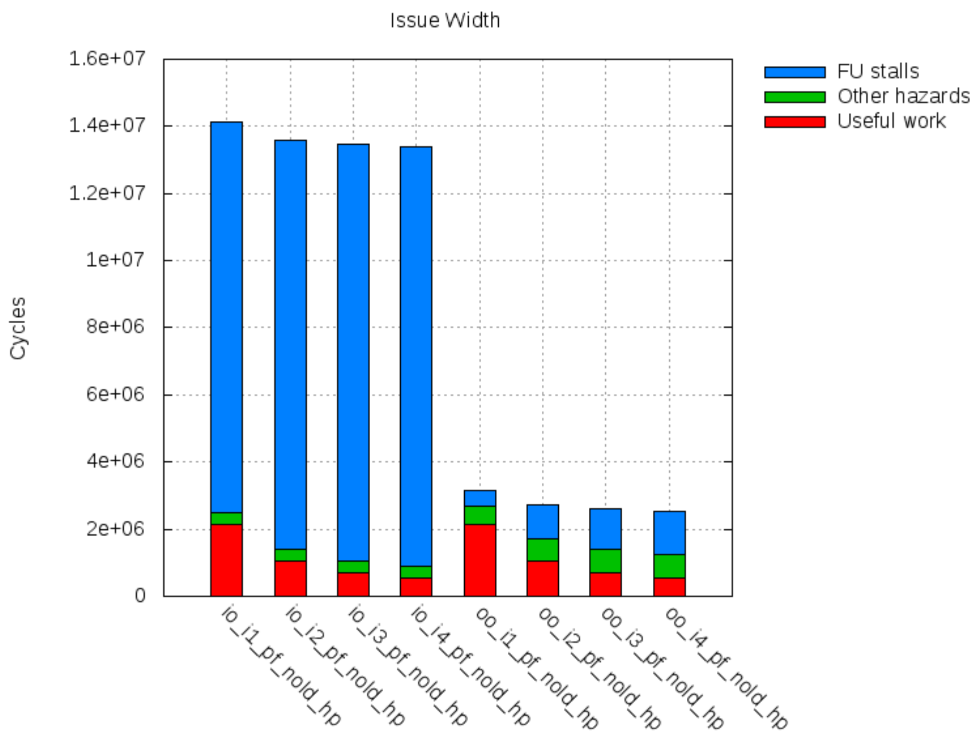

Figure 11: Gzip Results for Issue Width.

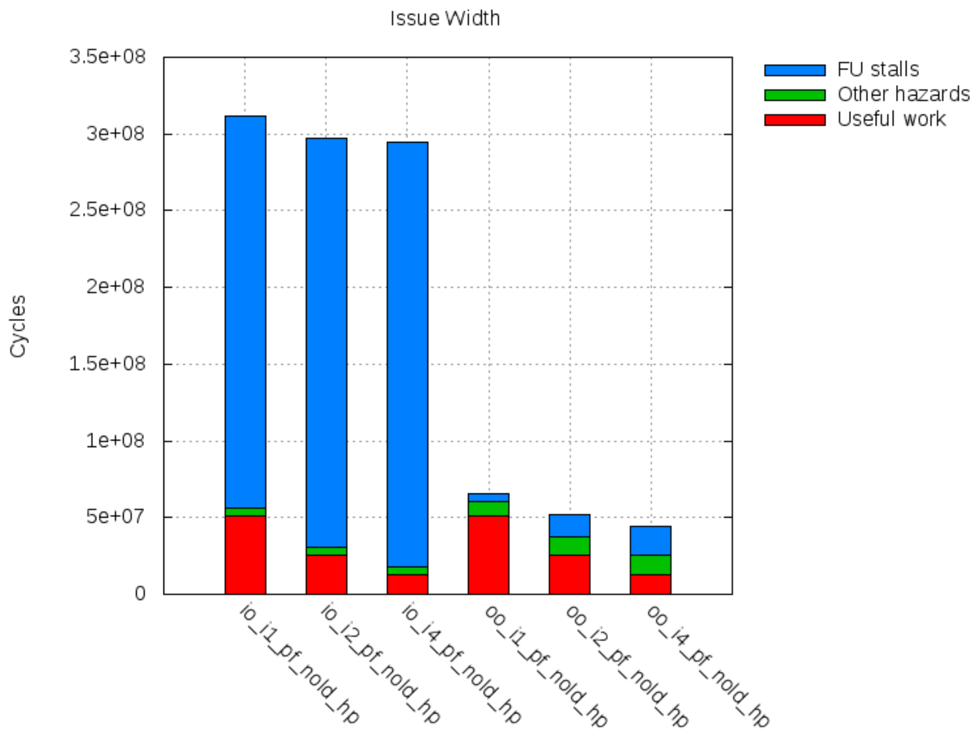

Figure 12: Oggenc Results for Issue Width.

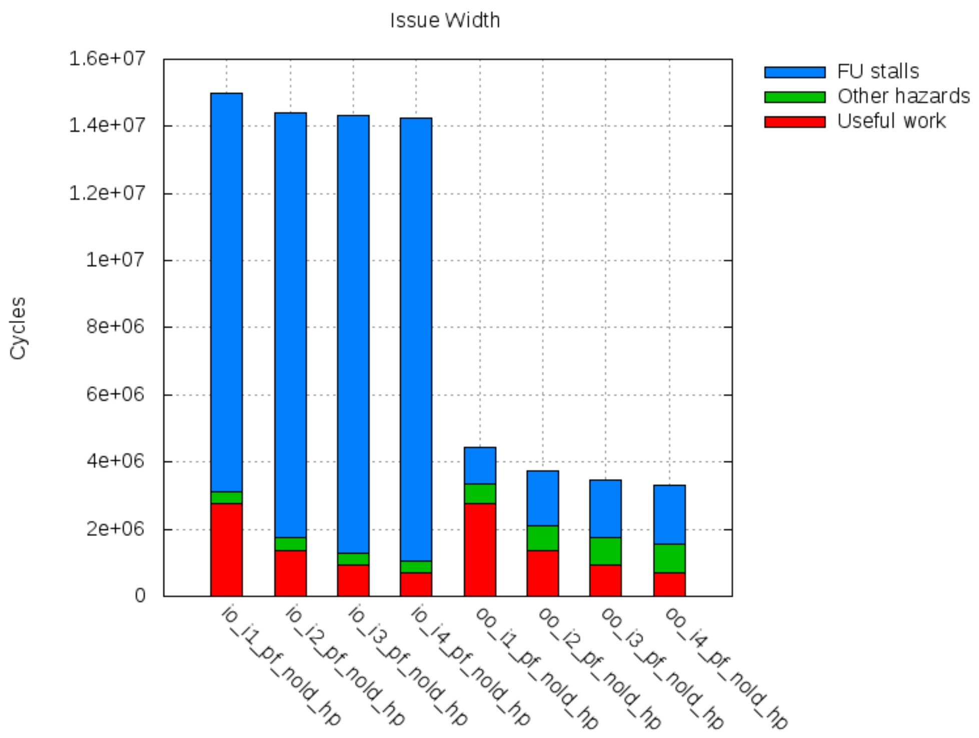

Figure 13: Blackscholes Results for Orderness.

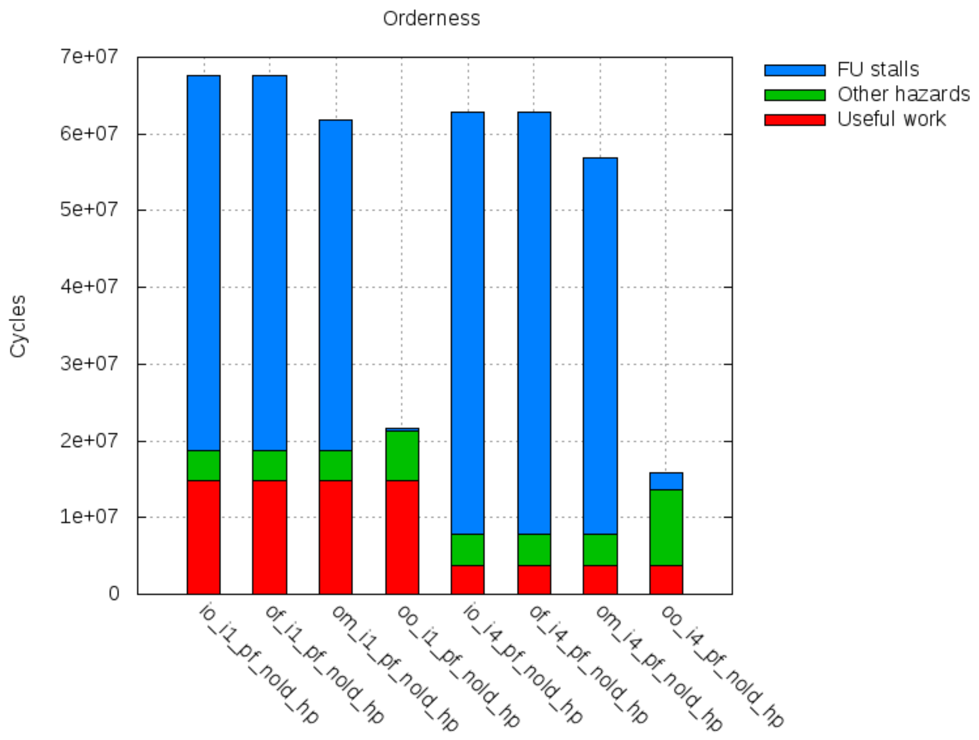

Figure 14: Picosat Results for Orderness.

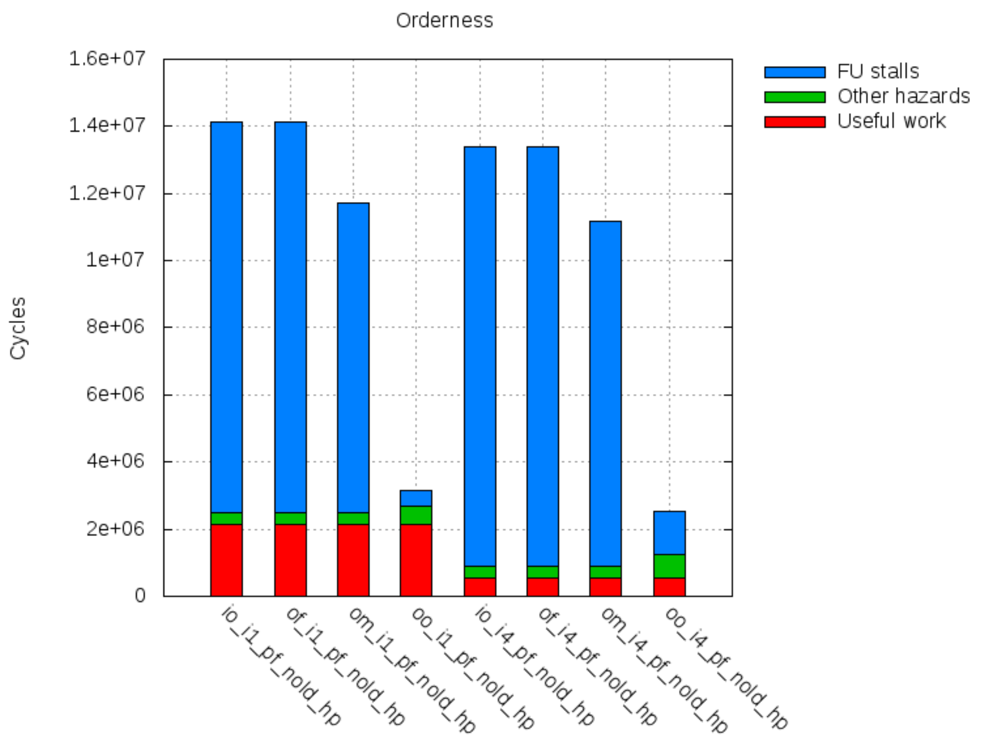

Figure 15: Gzip Results for Orderness.

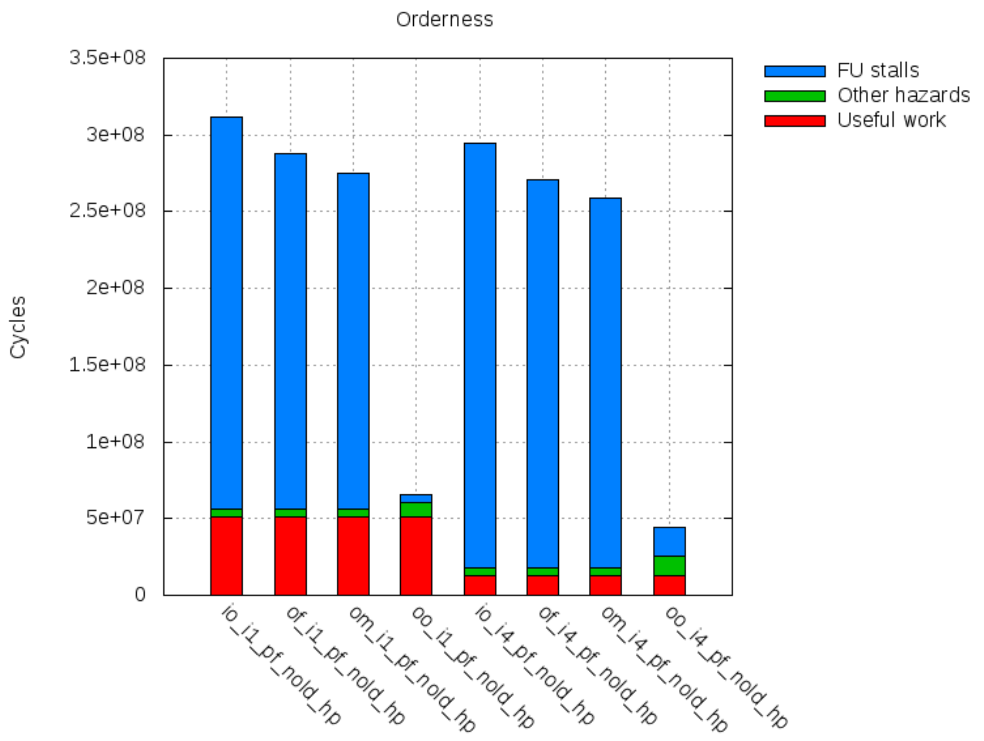

Figure 16: Oggenc Results for Orderness.

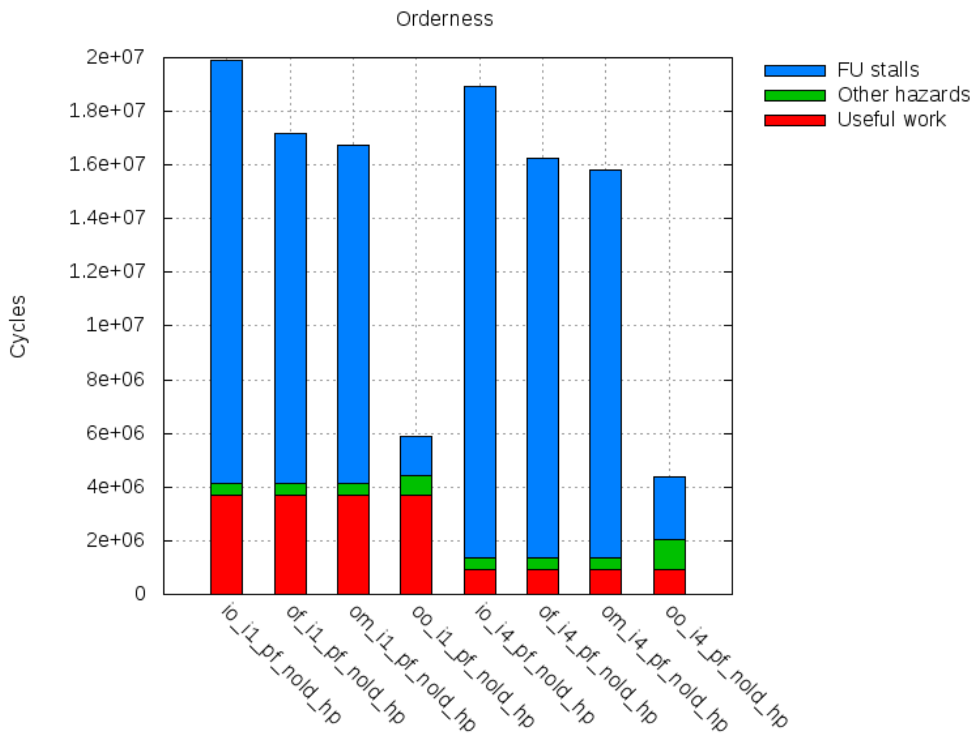

Figure 17: Blackscholes Results for Orderness.