

Feasibility of Combined Area and Performance Optimization for Superscalar Processors Using Searching

Sven van Haastregt¹ and Peter M.W. Knijnenburg²

¹ LIACS, Leiden University, The Netherlands
svhaastr@liacs.nl

² CSA, Informatics Institute
University of Amsterdam, The Netherlands
peterk@science.uva.nl

Abstract. When designing embedded systems, one needs to make decisions concerning the different components that will be included in a microprocessor. An important issue is the chip area vs. performance trade-off. In this paper we investigate the relationship between chip area and performance for superscalar microprocessors. We investigate the feasibility to obtain a suitable configuration by searching. We show that our approach gives a good configuration after 100 to 150 iterations using a simple random search algorithm. This shows the feasibility of our approach, in particular when more sophisticated search algorithms are employed as we plan in future work.

1 Introduction

Current embedded systems require high performance. Therefore, several current and many future embedded processors are out-of-order or even simultaneous multithreaded [9]. A drawback of these types of processor is that they consume much silicon area because of the complicated control structures required to support out-of-order execution [7]. This may be problematic for embedded systems where silicon area is expensive. Therefore, it is important to tune the architecture in such a way that maximum performance is achieved using a minimal amount of resources. Obviously, this is very difficult for general purpose processors, but in the case of embedded processors that only run a limited set applications, it may be possible to select a restricted set of resources in such a way that high performance still is achieved. For in-order processors there exist many approaches to explore the design space [5]. For example, PICO is an automatic system to explore application specific VLIW processors [8]. In the Artemis project [13] two different frameworks for the simulation phase are adopted: Spade [10] provides a model for rapid high level architecture performance simulations, and Sesame [14], provides a method for evaluating designs at multiple abstraction levels.

Recently, Eyerman et al. have proposed methods to explore the design space of out-of-order processors [4] focussing mainly on energy. In this paper, we study the feasibility to automatically search for good out-of-order processors configurations for specific applications, paying attention to both performance and area. That is, we want to find a processor configuration that is small but powerful enough for specific applications.

We use the SimpleScalar toolset as the design space to explore [1]. SimpleScalar allows us to set many architectural parameters. Each component has a different effect on the final performance. Furthermore, there exist dependencies between the several components. For example, increasing the number of arithmetical units will not increase performance, unless multiple instructions can be executed in parallel. It requires quite some analysis to find all dependencies between the various components and the list of dependencies quickly becomes complex. As we show in Section 2, even with a relatively small amount of possible design options (from now on referred to as *tuning parameters*), the search space is huge. Therefore, we employ a random search algorithm to explore only a fraction of this space. We show that in this way, using only about 100 to 150 configurations, we can find a high performance architecture that is much smaller than a general purpose architecture. This shows the feasibility of our approach, particularly when more sophisticated search algorithms would be developed as we plan in future work.

This paper is structured as follows. In Section 2, we describe the experiments we have performed with this new approach, and Section 3 contains the results of these experiments and we also give a short discussion. In Section 4, we mention some possible directions for future work. Section 5 summarizes this paper.

2 Experimental Setup

In this section, we discuss how we generate configurations, how performance is measured, the parameters of our experiments and the area model that is being used.

The search algorithm we use in our experiments is the most basic one available: we randomly generate a set of 1000 configurations (without duplicates) using different tuning parameters and then measure the performance and calculate the area of each configuration.

To evaluate the performance of each configuration, we use the *SimpleScalar Tool Set* [1]. The SimpleScalar simulator supports several instruction set architectures. We use the PISA architecture.

We use two applications for our experiments, *jpeg* and *mpeg2dec*. Both of these programs rely heavily on integer calculations and scarcely on floating point operations. Therefore, we keep the number of floating point arithmetical units constant throughout the experiments. The *jpeg*-simulation accounts for a total of about 1.1×10^9 instructions. The *mpeg2dec*-simulation results in about 1.3×10^8 instructions.

We have selected the following tuning parameters. In an iteration a value from the matching parameter value set is assigned to each parameter.

- **Data cache size:** number of bytes of the first level direct mapped data cache, block-size of 32 bytes. We use 6 values: { 1024, 2048, 4096, 8192, 16384, 32768 }
- **Instruction cache size:** number of bytes of the direct mapped instruction cache, blocksize of 32 bytes. We use 6 values: { 1024, 2048, 4096, 8192, 16384, 32768 }
- **GShare branch predictor size:** A GShare branch predictor consists of a w bits wide shift register (the global history register, containing the history of the w most

- recently executed branches) and a table containing 2^w bimodal counters [11]. We use 5 values: { 512, 1024, 2048, 4096, 8192 }
- **Branch Target Buffer (BTB) size:** the maximum number of entries in the BTB. We use 6 values: { 1, 64, 128, 256, 512, 1024 }
 - **Register Update Unit (RUU) size:** the number of slots available in the RUU, the unit that controls the out-of-order execution. We use 7 values: { 2, 4, 8, 16, 32, 64, 128 }
 - **Number of integer ALUs:** the number of integer Arithmetic Logic Units available. We use 5 values: { 1, 2, 3, 4, 5 }
 - **Number of memory ports:** the number of ports available to the CPU to access the first level cache. We use 4 values: { 1, 2, 3, 4 }
 - **Instruction fetch queue size:** the maximum number of instructions that can be stored in the fetch queue. We use 5 values: { 1, 2, 4, 8, 16 }
 - **Instruction issue width:** the maximum number of instructions that can be issued per cycle. We use 3 values: { 2, 4, 8 }
 - **Load/Store Queue (LSQ) size:** The LSQ handles the actual memory communication and contains a mechanism that avoids data hazards. We use 4 values: { 2, 4, 8, 16 }

All other possible architecture parameters remain constant throughout the experiment and are set at the SimpleScalar default values. With this set of parameters, more than nine million different configurations are possible.

To obtain an estimate of the area of a particular processor configuration, we use a slightly extended version of the model proposed by Steinhaus et al. [15]. This model provides an area estimate for a superscalar microprocessor design, specified using a SimpleScalar configuration, using analytical and empirical models. Chip area is expressed in λ^2 in order to get a quantity that is independent of the technology used to manufacture the microprocessor. Here, λ is defined as half of the minimum *feature size* which is the size of the smallest transistor, interconnect, etc. that can be produced by using a certain manufacturing process.

3 Results

In this section, we first show performance versus area for 1000 randomly picked parameter settings for two benchmarks, namely *jpeg* and *mpeg2dec*. Next, we show how much performance we obtain when we have area constraints.

3.1 Simulation Results

After running 1000 performance simulations for *jpeg* and *mpeg2dec*, we produced the plots in Figures 1 and 2. The *x*-axis represents the area corresponding to a single configuration and the *y*-axis shows the performance, which is calculated by:

$$\text{performance} = \frac{1}{\text{number of cycles}}$$

We have normalized the results to the SimpleScalar default configuration, given in Table 2. We also executed four additional simulations for each benchmark, which are plotted using horizontal lines. First, we determined the performance for the minimum and maximum configuration, by selecting the smallest and largest values, respectively, for each tuning parameter in our search space discussed in section 2. These are called “reachable minimum” and “reachable maximum”, respectively, in Figures 1 and 2. Next, we determined the absolute lower bound allowed by SimpleScalar by selecting the minimum value for each tuning parameter allowed by the SimpleScalar simulator. Finally, we determined an estimate of the upper bound by selecting very large values for each tuning parameter as listed in Table 1. The sizes of these configurations are listed in Table 3.

Register Update Unit size	2048 slots
Data cache size	16 Megabytes
Instruction cache size	16 Megabytes
GShare branch predictor size	524288 entries
Branch target buffer size	524288 entries
Number of integer ALUs	8 (maximum)
Number of memory ports	8 (maximum)
Instruction issue width	64 instructions per cycle
Instruction fetch queue size	64 instructions
Load/Store Queue size	1024 entries

Table 1: Parameter settings for the configuration that is our estimated upper bound.

Register Update Unit size	16 slots
Data cache size	4 Megabytes
Instruction cache size	16 Megabytes
GShare branch predictor size	2048 entries
Branch target buffer size	512 entries
Number of integer ALUs	4
Number of memory ports	2
Instruction issue width	4 instructions per cycle
Instruction fetch queue size	4 instructions
Load/Store Queue size	8 entries

Table 2: Parameter settings for the SimpleScalar default configuration

Figures 1 and 2 show that there is a difference in performance between the minimum reachable and maximum reachable configurations of about a factor of five. Compared to this, the difference between the reachable minimum and the SimpleScalar minimum is quite small. The same applies to the difference between the reachable maximum and

Configuration	Area (M λ^2)	Speedup	
		ijpeg	mpeg2
Reachable minimum	11250	1.0	1.0
Reachable maximum	44539	6.4	7.5
Minimal configuration	11168	0.6	0.7
Huge configuration	13764139	7.2	8.5

Table 3: Area and Speedup wrt minimum configuration of reference configurations.

the large SimpleScalar configuration. Thus, the value sets we have chosen for the tuning parameters cover a broad range of the search space.

One immediately notices the four clusters that appear in both plots. These turn out to be caused by the “number of memory ports” parameter: each value for this parameter corresponds to a cluster. Since this parameter has a huge impact on the total area of a configuration, it clearly separates the different classes. This is caused by the amount of additional wiring and logic needed for each memory port. For example, when the number of memory ports is increased by one, the load/store queue requires at least one additional read and write port for each of its SRAM cells. This is because the LSQ must be able to serve an additional read or write operation during a single cycle. The area of several other components, like the register file, TLB and cache, is influenced in a similar manner. However, it seems there is not much to gain anymore when the number of memory ports is higher than 2.

We observe that the majority of the configurations is located below two times the performance of the reachable minimum. However, there are some differences when looking at certain individual configurations. Some have a high performance for the `ijpeg` benchmark while that same configuration does not perform as well as in the `mpeg2dec` simulation, although the performance still lies above the average. Interestingly, this hardly holds conversely: configurations that perform well for the `mpeg2dec` benchmark are also among the best performing configurations of `ijpeg`.

3.2 Improvement under Area Restrictions

In this section, we study how fast the random search algorithm finds a good configuration when we impose a limit on the allowable area. Such a limit is important in practice when a system needs to be fit on a given amount of silicon. The measure of “goodness” we employ in this paper is speedup over the reachable minimum configuration. For a configuration x , $speedup(x)$ is given by:

$$speedup(x) = \frac{performance(x)}{performance(\text{min config.})}$$

Speedups of the reference configurations are shown in Table 3. We use area restrictions ranging from 12,000 to 30,000 M λ^2 . The resulting plots, shown in Figures 3 to 8, are produced by iterating over the set of 1000 configurations. The performance of each

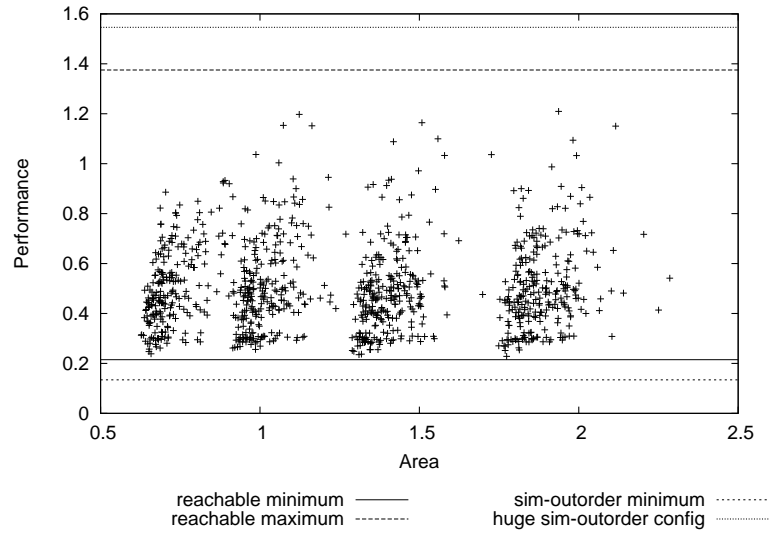


Fig. 1: Area vs. performance ijpeg, normalized wrt default configuration

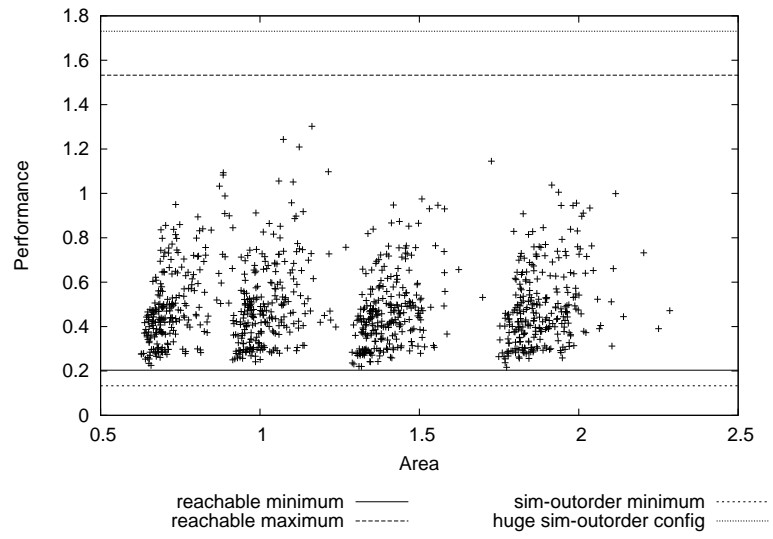


Fig. 2: Area vs. performance mpeg2dec, normalized wrt default configuration

configuration that satisfies the area restriction is plotted. The two different lines in a figure indicate the best configuration encountered so far for both benchmarks.

In Figure 3 we pose a limit that is only slightly larger than the minimum area shown in Table 3. We still produce a configuration that is almost twice as fast as this minimal one. This shows that carefully selecting a few extra resources can be highly effective.

Limits of 13,000 to 15,000 $M\lambda^2$ produce better configurations with speedups of around 4. For `jpeg` these limits deliver the same configurations, as shown in Table 4. For `mpeg2dec`, a larger value for the limit is used for a larger instruction cache, as shown in Table 4. This indeed gives a higher performance, as shown in Figures 5 and 6.

When the limit allows more than 1 memory port, 2 memory ports are chosen, as shown in Table 3. Figures 7 and 8 show that this gives more performance than 1 port. However, when the limit is 30,000 $M\lambda^2$, 3 ports could be accommodated. However, this value is not chosen, indicating that such extra port does not give extra performance compared to 2 ports.

Finally, we note that we only need around 100 simulations to find good candidates, irrespective of the limit we impose on the area. This shows that our simple approach of using a random search algorithm is already reasonably effective.

Combining the configurations in Table 4 and the speedups from Figures 3 to 8, it is clear that both caches do not need to be that large for the `jpeg` benchmark. A data cache of 2048 bytes and an instruction cache of 4096 bytes should be sufficient. The reason that data caches can be small lies in the algorithms used in this benchmark: many computations are in essence “local” because discrete transforms are applied to small 8×8 blocks. Also, the compute intensive loops are small so that for `jpeg` small caches can be sufficient. For the `mpeg2dec` benchmark, the same holds for the data cache, but the preferred instruction cache size turns out to be 32 kilobytes. This stresses that one should be careful when evaluating simulation data: the microprocessor configurations that are returned by our approach depend greatly on the benchmark applications used in the simulation step. It shows how important it is to choose the right benchmark suite when designing a microprocessor.

In general, the RUU size needs to be at least 32 and the BTB size at least 64. In the best performing configurations, the branch predictor size varies between the lowest and highest possible values. So it seems this parameter (or the value set we have chosen for it) does not have a big influence on the performance in our experiments. For the `jpeg` benchmark, the average branch predictor accuracy is about 89%. For the `mpeg2dec` benchmark, the average accuracy is about 97%. In general, the accuracy doesn't deviate more than 1% from the average for both benchmarks. The minimum number of integer ALUs that need to be included turns out to be three for both benchmark applications. The fetch queue size, issue width and load/store queue size tend to the higher values of the parameter set for a good performance result (≥ 4 , ≥ 4 , ≥ 8 respectively). The only thing in which both benchmarks significantly differ is the fetch queue size: in general the `mpeg2dec` benchmark performs slightly better when the fetch queue size equals eight or sixteen, compared to configurations that have a smaller fetch queue size.

data cache	instr. cache	branch pred.	BTB	RUU	#ALUs	#memports	FQsize	Issue width	LSQ
jpeg									
area \leq 12000 M λ^2									
2048	16384	512	512	16	1	1	2	2	4
area \leq 13000 M λ^2									
2048	8192	2048	512	32	5	1	4	4	16
area \leq 14000 M λ^2									
2048	8192	2048	512	32	5	1	4	4	16
area \leq 15000 M λ^2									
2048	8192	2048	512	32	5	1	4	4	16
area \leq 20000 M λ^2									
8192	32768	4096	64	64	3	2	8	4	16
area \leq 30000 M λ^2									
8192	16384	1024	256	128	4	2	4	8	16
mpeg									
area \leq 12000 M λ^2									
1024	2048	512	512	8	5	1	2	2	16
area \leq 13000 M λ^2									
2048	8192	2048	512	32	5	1	4	4	16
area \leq 14000 M λ^2									
2048	32768	1024	128	64	3	1	4	4	8
area \leq 15000 M λ^2									
2048	32768	1024	128	64	3	1	4	4	8
area \leq 20000 M λ^2									
8192	32768	4096	64	64	3	2	8	4	16
area \leq 30000 M λ^2									
2048	32768	2048	256	128	5	2	8	4	16

Table 4: Best Configurations found by random search when applying size constraints

4 Future Work

In this paper, we have used a very simple random search algorithm. The results of the simulations are not used for any feedback. Doing so could improve the search. For example, genetic algorithms can be used. Another direction is by applying data mining techniques on the obtained data, which consists of the configurations together with their estimated area and computed performance to create heuristics in order to decrease the size of the search space. An example heuristic can restrict the number of ALUs to the number of instructions that can be fetched simultaneously. Another heuristic can prevent a configuration from having more LSQ slots than RUU slots. Furthermore, one could try to improve the performance simulation step. A possible way to do this, is to use small, but representative inputs for the benchmark applications used in the simulations [3]. Another approach could use statistical simulation [12,2]. We can also apply Pareto-Front Arithmetics [6] to minimize the part of the design space to be evaluated.

5 Conclusion

In this paper we have demonstrated the feasibility of an iterative approach to the problem of finding suitable microprocessor configurations: we can find a high performance configuration that satisfies a given area restriction using a simple search algorithm and a limited number of iterations. We have shown that even a small increase in the resources compared to a minimal configuration can give a speedup of 2.5, which implies that tuning a processor can be highly effective. Our results suggest that around 100 evaluations could be sufficient. However, this can still be too time consuming, in particular when several applications need to be accommodated. Therefore, in future work, we focus on reducing this number by designing more sophisticated search algorithms than the random search from this paper.

References

1. T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.
2. L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proc. PACT*, 2001.
3. L. Eeckhout, H. Vandierendonck, and K. de Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *J. of Instruction-Level Parallelism*, 5:1–33, 2003.
4. S. Eyerman, L. Eeckhout, and K. De Bosschere. Efficient design space exploration of high performance embedded out-of-order processors. In *Proc. DATE*, 2006.
5. M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.
6. C. Haubelt and J. Teich. Accelerating design space exploration using pareto-front arithmetics. In *Proc. ASP-DAC*, pages 525–531, 2003.
7. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

8. V. Kathail, S. Aditya, R. Schreiber, B.R. Rau, D. Cronquist, and M. Sivaraman. PICO: Automatically designing custom computers. *IEEE Computer*, 35(9):39–47, 2002.
9. M. Levy. Multithreaded technologies disclosed at MPF. *Microprocessor Report*, November 2003.
10. P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proc. SiPS*, pages 181–190, 1999.
11. S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Lab., 1993.
12. S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Proc. PACT*, pages 15–24, 2001.
13. A. D. Pimentel, P. Lieverse, P. van der Wolf, L.O. Herzberger, and E. F. Deprettere. Exploring embedded-systems architectures with artemis. *IEEE Computer*, 34(11):57–63, 2001.
14. A.D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. on Computers*, 55(2):99–112, 2006.
15. M. Steinhaus, R. Kolla, J. Larriba-Pey, T. Ungerer, and M. Valero. Transistor count and chip-space estimation of simple-scalar-based microprocessor models. In *Proc. Workshop on Complexity-Effective Design*, 2001.

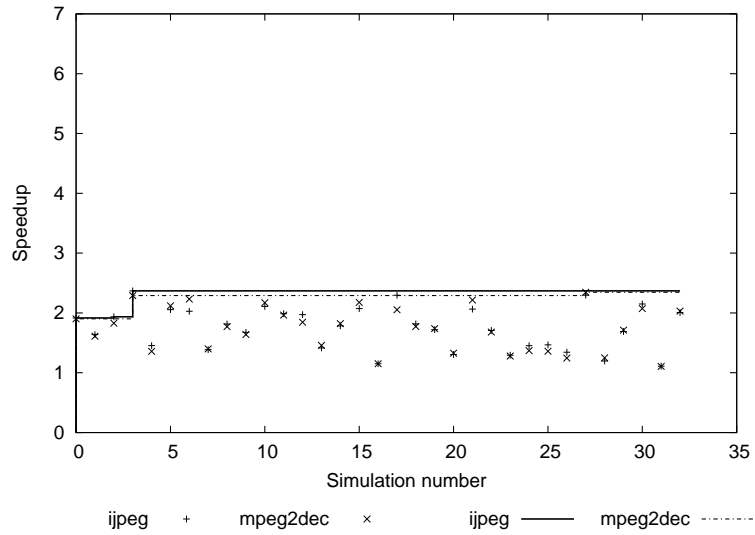


Fig. 3: Area $\leq 12000M\lambda^2$

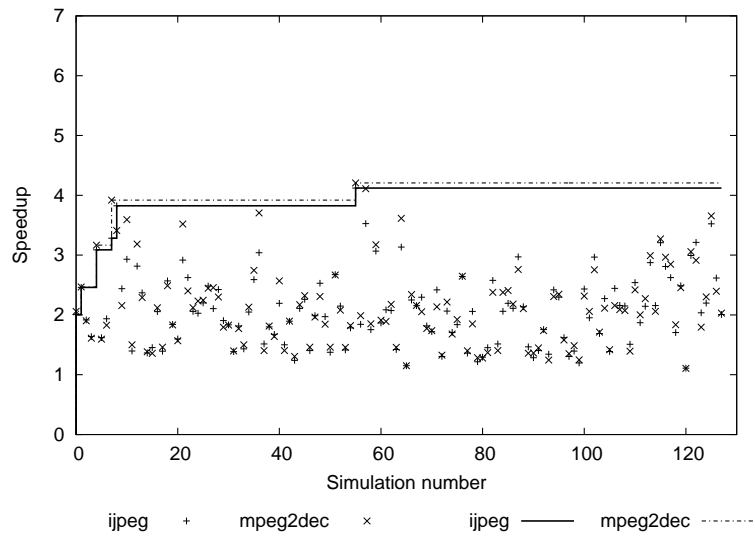


Fig. 4: Area $\leq 13000M\lambda^2$

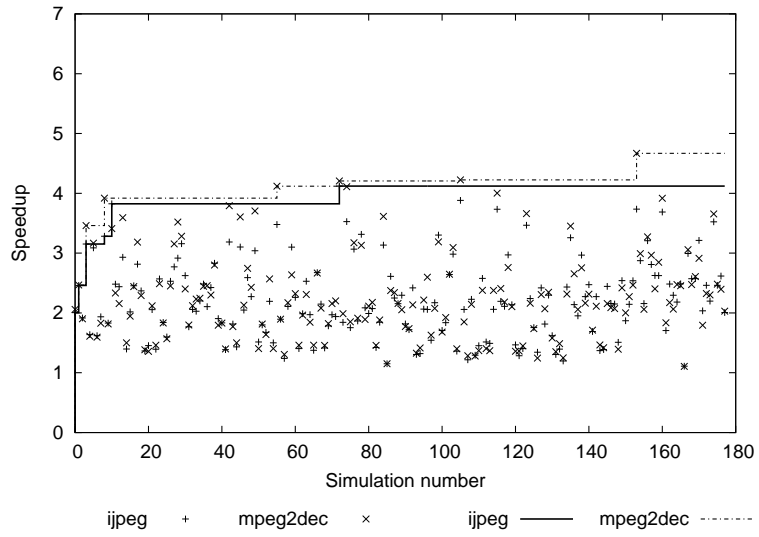


Fig. 5: Area $\leq 14000M\lambda^2$

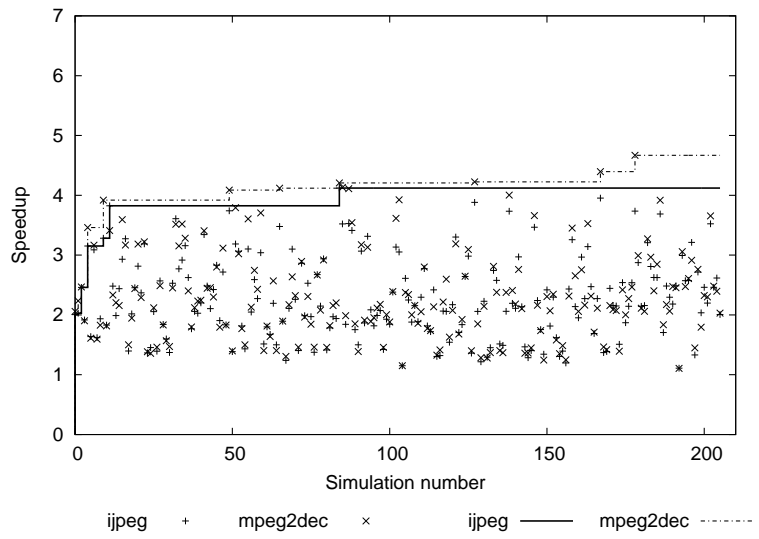


Fig. 6: Area $\leq 15000M\lambda^2$

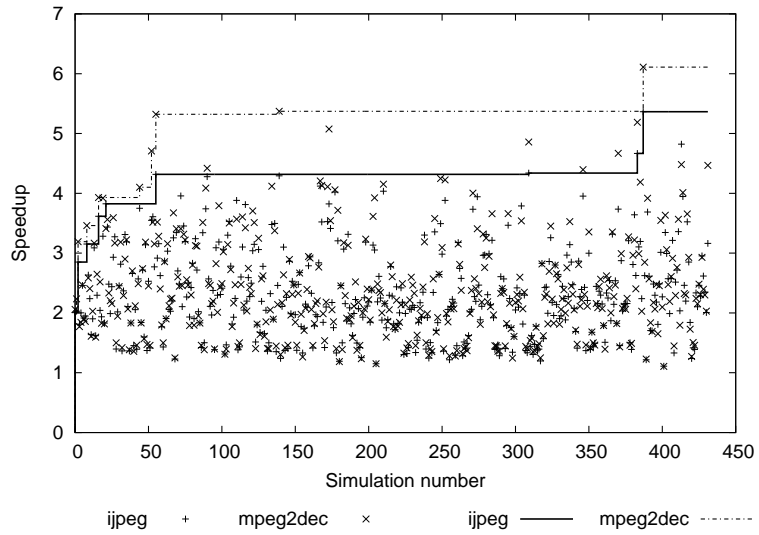


Fig. 7: Area $\leq 20000M\lambda^2$

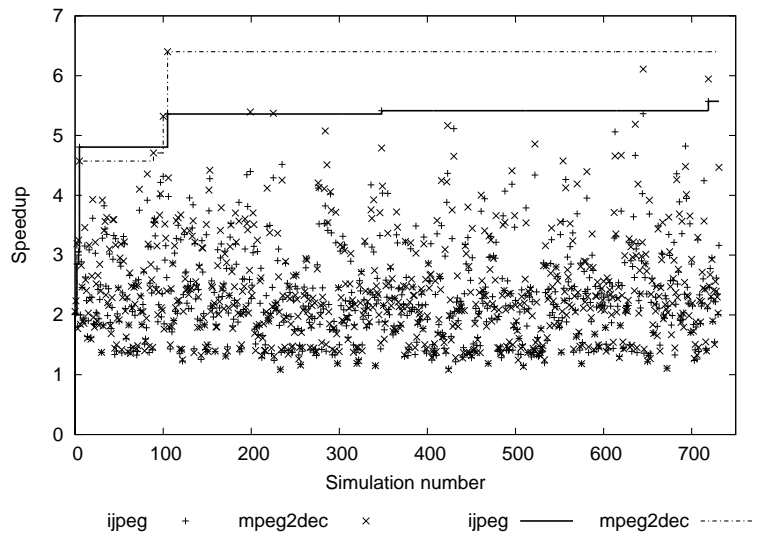


Fig. 8: Area $\leq 30000M\lambda^2$