An Iterative Approach to Area and Performance Optimization for Superscalar Processors

Sven van Haastregt
LIACS, Leiden University

14th September 2006

Abstract

When designing embedded systems, one needs to make decisions concerning the different components that will be included in a microprocessor. An important issue in this phase is the chip area vs. performance trade-off. In this paper we investigate the relationship between chip area and performance for superscalar microprocessors. We evaluate how one can obtain a suitable configuration by using an approach that is based on iterative compilation. It turns out that our approach returns a reasonably suitable configuration after a few hundred iterations. We investigate if this approach is feasible enough to be used in practice.

1 Introduction

Current embedded systems require high performance. Therefore, several current and many future embedded processors are out-of-order or even simultaneous multithreaded [1]. A drawback of these types of processors is that they consume much silicon area because of the complicated control structures required to support out-of-order execution [2]. For several reasons (both economical and practical) one always strives for a microprocessor that yields maximum performance, but has a small area. If there is plenty of space in the intended system, one might conclude that there is no urgent need to minimize the chip area. Unfortunately, often the available area *is* restricted due to cost constraints. Therefore, the designer needs to distribute the available area among the several components (like caches, branch predictor, arithmetical units, ...) in such a way that a maximum performance will be obtained. For general purpose processors, it is very difficult to find a suitable configuration. In the case of an embedded processor however, which only runs a limited set of applications, it may be possible to select a restricted set of resources in such a way that high performance is still achieved.

Furthermore, microprocessors with a large area might suffer from larger delays because it will take more time to send a signal from one location on the chip to another. Thus, designers could be forced to have the microprocessor run at

a lower clockspeed, in order to provide enough time for the transport of signals across the chip. In general, one wants to avoid these kinds of situations.

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. Therefore, it is quite possible that a human designer overlooks the effect of one or more interactions between certain components. In that case, a proposed configuration will probably not be an optimal one.

In this paper, we discuss a different approach for finding a suitable out-of-order processor configuration that is small, but powerful enough for specific applications. This approach, which is discussed in Section 2, is based on the technique of "iterative compilation" [4], in particular, the variant of iterative compilation in which randomness plays an important role. This makes an automatic search for good processor configurations possible. In Section 3, we describe the experiments we have performed with this new approach, and Section 4 contains the results of these experiments. In Section 5, we discuss these results. In Section 6, we mention some possible directions for future work and in Section 7, we discuss some related work that deals with design space exploration. Finally, Section 8 summarizes this paper.

2 Background

As we show in Section 3.3, even with a relatively small amount of possible design options (from now on referred to as *tuning parameters*), the search space is huge. Evaluating all possible combinations of the tuning parameters accurately, by using cycle-by-cycle simulation with several target applications, would take about seven centuries for a single Pentium 4 at 2.8 GHz. So we obviously need another approach.

In the domain of compiler optimization, a technique called *iterative compilation* has been developed in order to find optimal values for various optimization parameters [3, 4]. This is done by selecting a value for each parameter, which can be done randomly or by using a more "intelligent" approach like a genetic algorithm. Some other possible approaches are discussed in [3]. Next, the iterative compilation algorithm compiles a source program with these parameter values, executes it and measures its execution time. This process is repeated a number of times, with or without using the measured execution time as feedback for the parameter selection. Finally, the set of parameter values that leads to the shortest execution time is returned. Iterative compilation performs surprisingly well: in a reasonable amount of time, a level of optimization is found that performs better than well-known analytical techniques do [5].

We want to apply this principle of iterative compilation, that is, *searching* for optimal parameter values instead of computing them by analytical means, to

the problem of selecting the most optimal parameter values for a processor configuration. Parameter value selection can be done in the same way, i.e., random, or by using a more sophisticated approach. The evaluation of a configuration consists of two parts. First, one has to obtain a performance measurement of the configuration. Fortunately, this can be done using software simulation, so it is not necessary to produce a real physical version of the proposed processor. Otherwise this approach would become quite expensive if one wants to perform more than a few iterations of the algorithm. Second, one has to obtain the physical area or at least a heuristic estimate of it. Again, the most accurate but also most expensive method is to actually manufacture each processor that the algorithm proposes. Obviously, this is too expensive and time consuming to be feasible. However, for our purpose, a model which estimates the area is sufficient.

3 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.

3.1 Search Algorithm

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.

In Section 4.3, we show how the search algorithm can be modified, by performing the simulation step only when a certain area constraint is satisfied.

3.2 Performance Simulations

To evaluate the performance of each configuration, we use the *SimpleScalar Tool Set* [7]. This tool set offers several simulators, ranging from a simple functional simulator to a detailed out-of-order issue superscalar processor simulator. The latter includes (among other features) support for two cache layers and a branch predictor. We use the out-of-order simulator, since that simulator offers the largest amount of configurable processor parameters.

The SimpleScalar out-of-order simulator (`sim-outorder`) first takes a series of architecture parameters and a binary executable (together with a command line for this executable) as input. Next, it runs the application on a virtual out-of-order issue superscalar processor. This is achieved by simulating each individual cycle, thus maintaining the state of each component during the execution of the application. Hence, performing a simulation is a very compute intensive job.

After the executable has been simulated, SimpleScalar reports several execution statistics. These include the total simulation time in cycles, in which we are interested most.

The SimpleScalar simulator supports several instruction set architectures. We use the PISA architecture, which is similar to a MIPS instruction set architecture. Our main reason for this choice is the fact that the PISA-support is already part of the SimpleScalar package. Furthermore, PISA binaries for the intended benchmarks appear to be easier to obtain.

When performing a simulation, SimpleScalar executes the code of a binary executable that is compiled for the appropriate instruction set architecture. We use two applications for our experiments, `ijpeg` and `mpeg2dec`. The first is part of the SPEC CINT95 Benchmark collection [8] and performs compression and decompression of in-memory images. The latter is developed by the MPEG Software Simulation Group [9] and decodes an MPEG-2 movie. 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 ijpeg-simulation accounts for a total of about 1.1×10^9 instructions. The mpeg2dec-simulation results in about 1.3×10^8 instructions.

3.3 Parameters

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

- **Register Update Unit (RUU) size**: the number of slots available in the RUU, the unit that controls the out-of-order execution. The RUU system combines register renaming, reservation stations and reorder buffers into a centralized structure. For each instruction, a new entry (containing information like associated functional unit, source and destination operand contents and status data) is put into a free slot of the RUU. In general, this entry persists until the instruction commits. During its existence, the RUU entry is being updated whenever relevant data (e.g., source operand values or execution results) becomes available or status changes occur.
  We use 7 values: { `2, 4, 8, 16, 32, 64, 128` }

- **Data cache size**: the size, in bytes, of the first level data cache. We use a direct mapped cache, with a blocksize of 32 bytes.
  We use 6 values: { `1024, 2048, 4096, 8192, 16384, 32768` }

- **Instruction cache size**: the size, in bytes, of the first level instruction cache. We use a direct mapped cache, with a blocksize of 32 bytes.
  We use 6 values: { `1024, 2048, 4096, 8192, 16384, 32768` }

- **GShare branch predictor size**: the branch predictor enables speculative execution, thereby contributing to the efficiency of out-of-order execution. We use GShare, a global branch prediction scheme [10]. 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. To obtain a prediction for a branch, a table entry is chosen by taking (some bits of) the branch address, XORed with the global history register. The size of the branch predictor is specified by stating the number of entries in the table.
We use 5 values: { `512, 1024, 2048, 4096, 8192` }

- **Branch Target Buffer (BTB) size**: the maximum number of entries in the BTB. The BTB stores the predicted address for the next instruction that has to be fetched after a branch. With a BTB, the processor is able to fetch the next instruction at an earlier stage in the pipeline and hence, the branch penalty can be reduced.
  We use 6 values: { `1, 64, 128, 256, 512, 1024` }

- **Number of integer ALUs**: the number of integer Arithmetic Logic Units available. The more ALUs available, the more arithmetical and logical operations can be done in parallel (provided that no data hazards occur; this is handled by the out-of-order control logic).
  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. With additional memory ports, the system memory can be accessed simultaneously by multiple functional units.
  We use 4 values: { `1, 2, 3, 4` }

- **Instruction issue width**: the maximum number of instructions that can be issued (i.e., moved from the instruction decode stage to the execution stage) during a cycle.
  We use 3 values: { `2, 4, 8` }

- **Instruction fetch queue size**: the maximum number of instructions that can be stored in the fetch queue. During the instruction fetch stage, instructions are transferred from the instruction cache or the system memory into the fetch queue.
  We use 5 values: { `1, 2, 4, 8, 16` }

- **Load/Store Queue (LSQ) size**: the LSQ supports the RUU during all load and store instructions (i.e., instructions that access memory). The RUU still handles execution control and performs the effective address calculation. 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, about nine million different configurations are possible.

3.4 Area Model

To obtain an estimate of the area of a particular processor configuration, we use a slightly extended version of the model proposed by Marc Steinhaus et al. [6]. This model provides an area estimate for a superscalar microprocessor design, specified using a SimpleScalar configuration. Using analytical and emperical models, the number of transistors and the chip area is estimated. 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 using a certain manufacturing process).

Unfortunately, the model only provides an estimate for bimodal branch predictors, while we use a GShare branch predictor. Therefore we have extended the model. For the exact details concerning the several assumptions and formulas we refer to [6]. However, our explanation about the extension of the model should provide enough detail to get a satisfying understanding of the entire model.

Essentially, the GShare branch predictor is just another set of (multiport) SRAM cells, much like a Branch Target Buffer. For simplicity, we ignore the small amount of additional control logic required. Thus extending the model is done by applying the same techniques that are used for this component, with the difference that the GShare predictor consists of one table and a history register.

To obtain the entire area of a branch predictor, the number of bits used in the predictor is multiplied by the area (which equals height times width) of a single SRAM cell:

$$AreaBpred = BitPerBpred \cdot BpredCellHght \cdot BpredCellWid$$

The variable $BitPerBpred$ specifies the number of bits contained in the entire predictor. The number of bits in a GShare predictor, with a shift register width of W bits and 2^W entries in the bimodal table, can be expressed by:

$$BitPerBpred = W + 2 \cdot 2^W = W + 2^{W+1}$$

Next, we need to calculate the dimensions $BpredCellWid$ and $BpredCellHght$ of a single SRAM cell. In the model we use, a SRAM cell without any wires connected to it has a basic width and height, called $SRCellBasicWidInLam$ and $SRCellBasicHghtInLam$, respectively. Two types of wires have to be connected: data wires, which transfer the data that has been read or has to be written, and address wires, which signal whether the cell is selected for an operation or not. Steinhaus et al. [6] have chosen to connect all address wires to the smaller side of the cell, that is, connect them to the side designated by $BpredCellHght$, and all data wires to the broader side (i.e., $BpredCellWid$). The number of address and data wires depends on the number of read and write ports. Both a read port and a write port require one address wire, so the height of a single SRAM cell of the GShare branch predictor equals:

$$BpredCellHght = SRCellBasicHghtInLam + (BpredReadPort + BpredWrtPort) \cdot WtWidInLam$$

Nearly the same applies to the width of a single SRAM cell, except that we should look at data wires instead of address wires. A read port requires one data wire, while a write port requires two data wires. The leads to:

$$BpredCellWid = SRCellBasicWidInLam + (BpredReadPort + 2 \cdot BpredWrtPort) \cdot WtWidInLam$$

In both formulas, $WtWidInLam$ represents the width in λ needed for a single data or address wire. $BpredReadPort$ and $BpredWrtPort$ represent the number of read and write ports of the SRAM cell, respectively. Since the maximum number of read and write ports required during a cycle depends on the maximum number of instructions that can be fetched simultaneously, it is made equal to the instruction fetch width:

$$BpredReadPort = BpredWrtPort = FetchWidth$$

With this extension added to the original model, we obtain a suitable area estimation model for our experiment.

The authors of the original area model also developed a Microsoft Excel spreadsheet which allows one to enter the various SimpleScalar configuration parameters. All calculations are then performed automatically and one can immediately view the estimations of transistor count and chip area.

In order to get a tool which is more practical in an iterative search Unix-environment, we developed a small C-program that performs these estimations. This tool, called `sim-area`, has roughly the same commandline interface as the original SimpleScalar simulator: all configuration parameters that appear in `sim-outorder` (the out-of-order execution simulator of SimpleScalar), are also supported by our tool using exactly the same syntax.

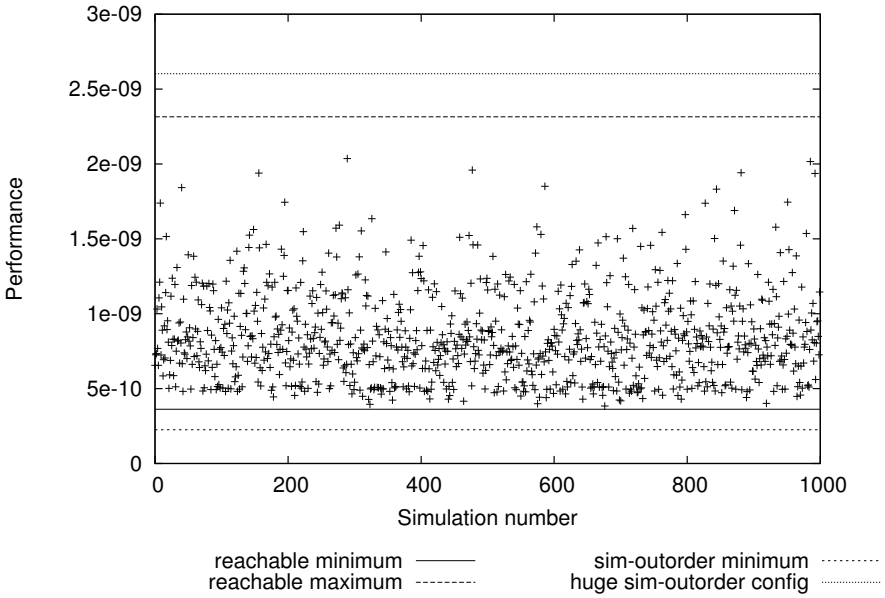
4 Results

In this section, we show the results of the experiments we described in the previous section.

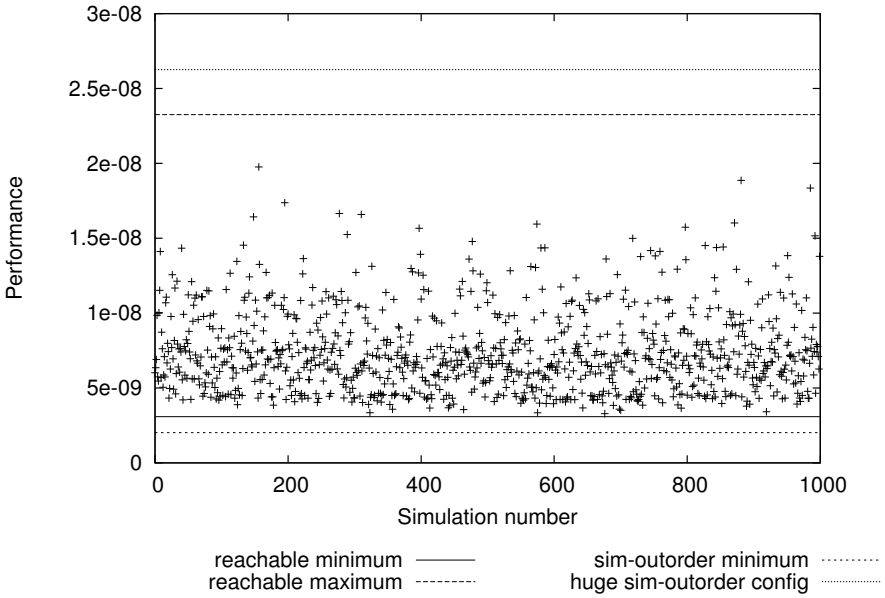
4.1 Simulation Results

First, we generated 1000 unique parameter sets. After running 1000 performance simulations for both the `ijpeg` benchmark and the `mpeg2dec` benchmark, we produced the plots of Figure 1. The x-axis represents the simulation number, which correspronds to a single configuration. The y-axis shows the performance, which is calculated by:

$$\text{performance} = \frac{1}{\text{number of cycles needed for the simulation}}$$



(a) ijpeg



(b) mpeg2dec

Figure 1: Distribution of performance simulations

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. These are called "reachable minimum" and "reachable maximum". Next, we determined the absolute lower bound allowed by SimpleScalar by selecting the minimum value for each tuning parameter. Finally, we determined an estimate of the upper bound by selecting huge values for each tuning parameter. These values are listed in Table 1.

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 (SimpleScalar maximum)
Number of memory ports:	8 (SimpleScalar maximum)
Instruction issue width:	64 instructions per cycle
Instruction fetch queue size:	64 instructions
Load/Store Queue size:	1024 entries

Table 1: Parameters for an estimate of the upper bound.

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

Both plots show quite an identical pattern, with the majority of the configurations located below two times the performance of the reachable minimum. However, there are some differences when looking at certain individual configurations. Some (e.g., configurations 1 and 2 in Table 2) 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` (e.g., configurations 3 and 4 of Table 2). For the particular configurations given in this table, it seems the lower performance of configurations 1 and 2 for the `mpeg2dec` benchmark is caused by their smaller instruction cache. However, without any further investigation, one may not conclude this is true for all configurations.

Parameter	Conf. 1	Conf. 2	Conf. 3	Conf. 4
Register Update Unit size	64	32	128	64
Data cache size (bytes)	16384	32768	8192	8192
Instruction cache size (bytes)	8192	4096	16384	32768
GShare branch predictor size	512	4096	1024	4096
Branch target buffer size	1024	128	256	64
Number of integer ALUs	3	4	4	3
Number of memory ports	4	3	2	2
Instruction issue width	8	8	8	4
Instruction fetch queue size	4	8	4	8
Load/Store Queue size	16	16	16	16
Area ($M\lambda^2$)	35166	27382	20393	19491
`ijpeg` rank	1st	3rd	2nd	4th
`mpeg2dec` rank	12th	15th	3rd	2nd

Table 2: Four parameter configurations.

4.2 Area and Performance

Next, we computed the area of each configuration and generated the plots shown in Figure 2. In both plots, the y-axis ranges from the performance of the reachable minimum to the performance of the reachable maximum. The most interesting processor configurations (i.e., processor configurations that perform well and have a small area) appear in the upper left corner of the plot, while the lower right corner represents the less interesting configurations that do not perform well and require a considerable amount of area at the same time.

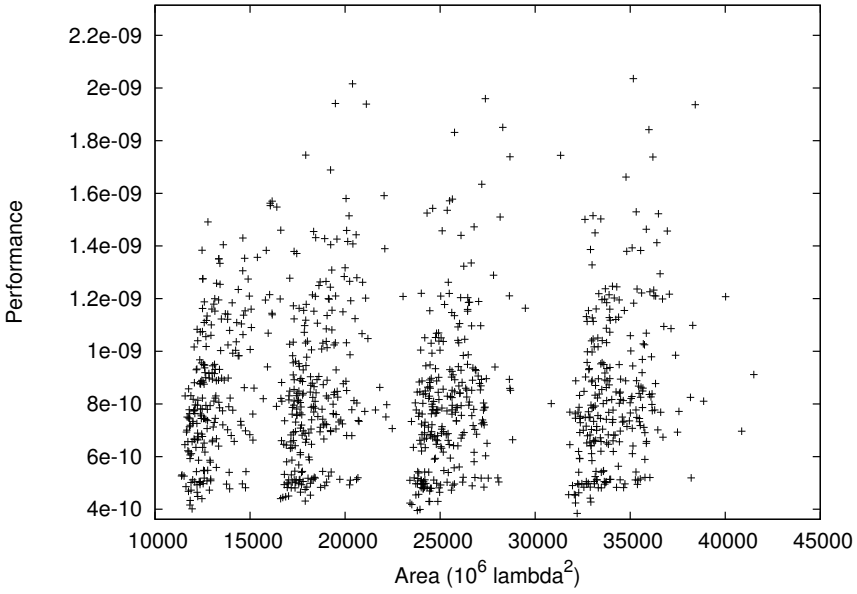
One immediately notices the four clusters that appear in both plots. These turn out to be caused by the "number of memory ports" parameter. Each possible value for this parameter corresponds to a cluster. Since this parameter has a huge impact on the total area of a configuration, it clearly separates the different classes. Again note that Figure 2(b) shows less high-performance "peaks" than Figure 2(a) does, especially when chip area exceeds 22000 $M\lambda^2$.

4.3 Gain under Area Restrictions

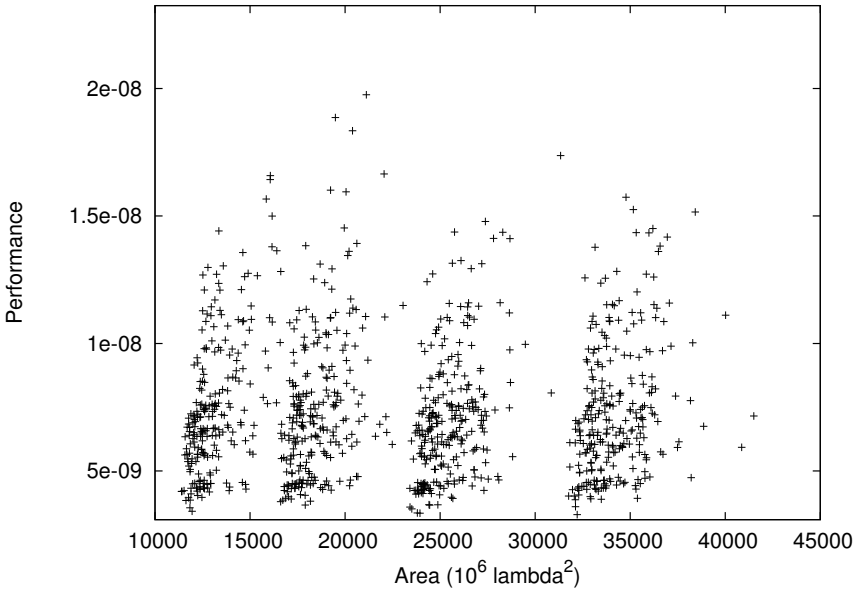
Finally, we discuss the performance of our iterative approach. Therefore, we plotted several graphs in Figure 3, with area restrictions ranging from 12000 to 30000 $M\lambda^2$. In order to compare both the `ijpeg` and `mpeg2dec` results in one figure, the y-axis contains the "speedup" relative to the minimum configuration. For a configuration x, this speedup is calculated by:

$$speedup(x) = \frac{performance(x)}{performance(\text{minimum configuration})}$$

A configuration with exactly the same performance as the minimum configuration has a speedup of 1, a configuration that performs twice as well as the

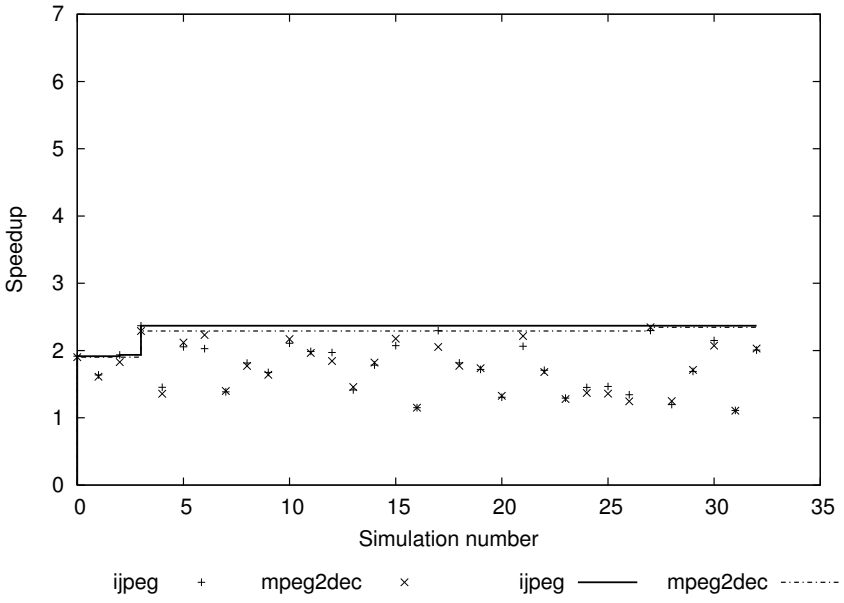


(a) ijpeg

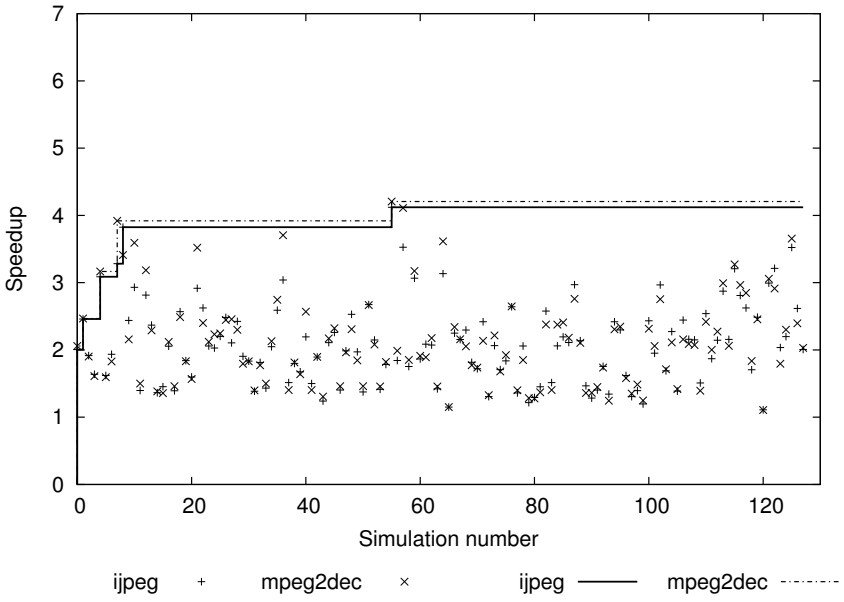


(b) mpeg2dec

Figure 2: Area and performance of all simulations

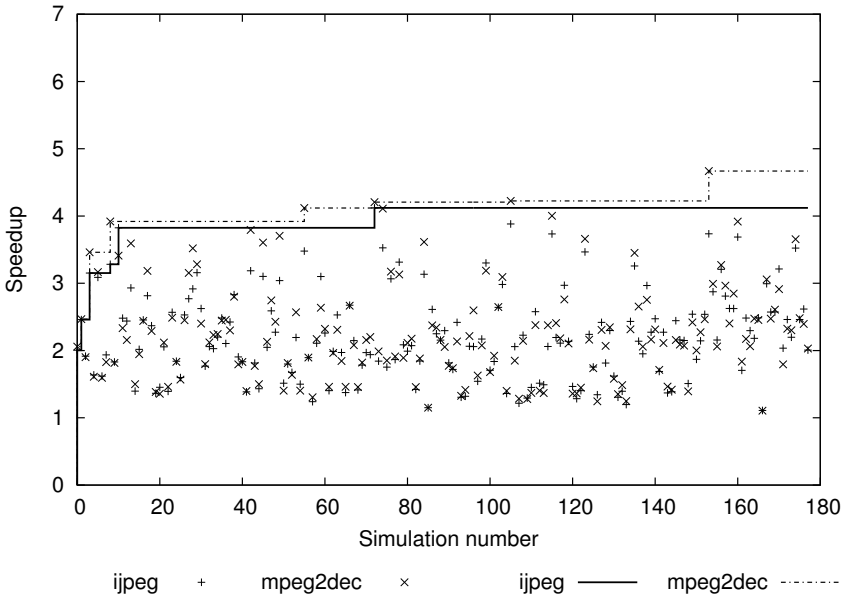


(a) area $\leq$ 12000 Mλ^2

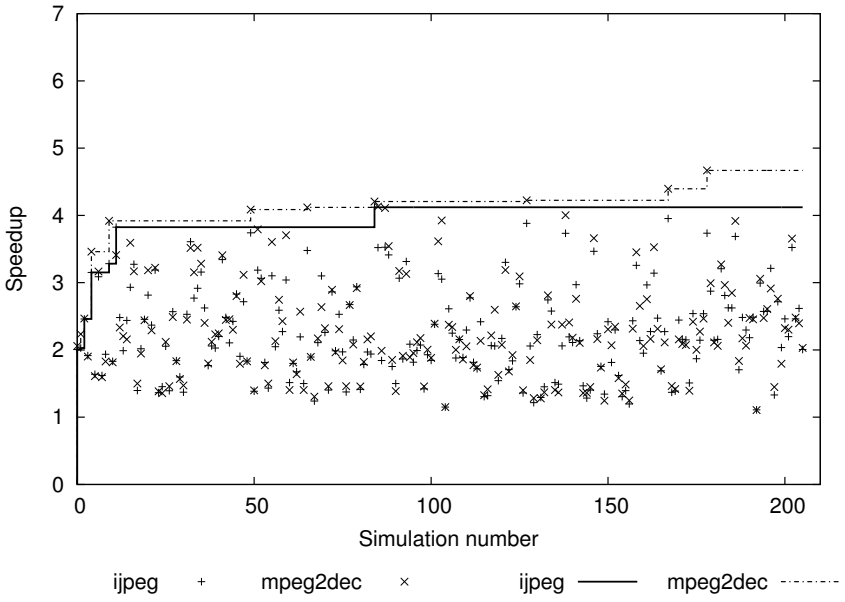


(b) area $\leq$ 13000 Mλ^2

Figure 3: Plots with various limits on area

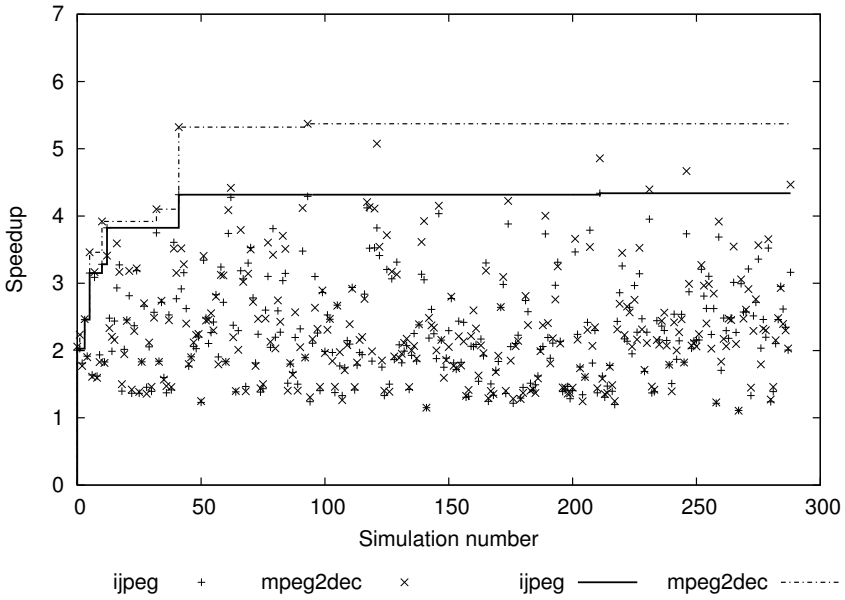


(c) area $\leq$ 14000 Mλ^2

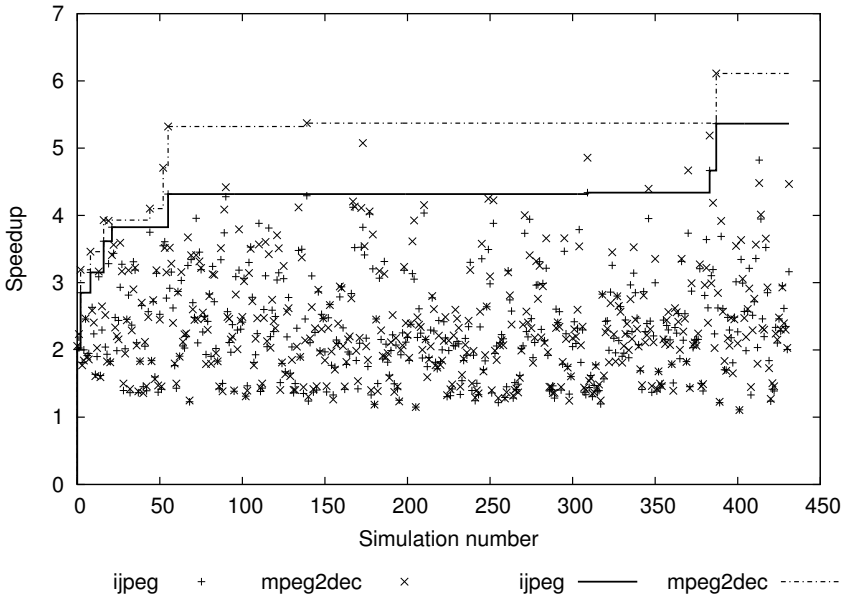


(d) area $\leq$ 15000 Mλ^2

Figure 3: Plots with various limits on area (continued)

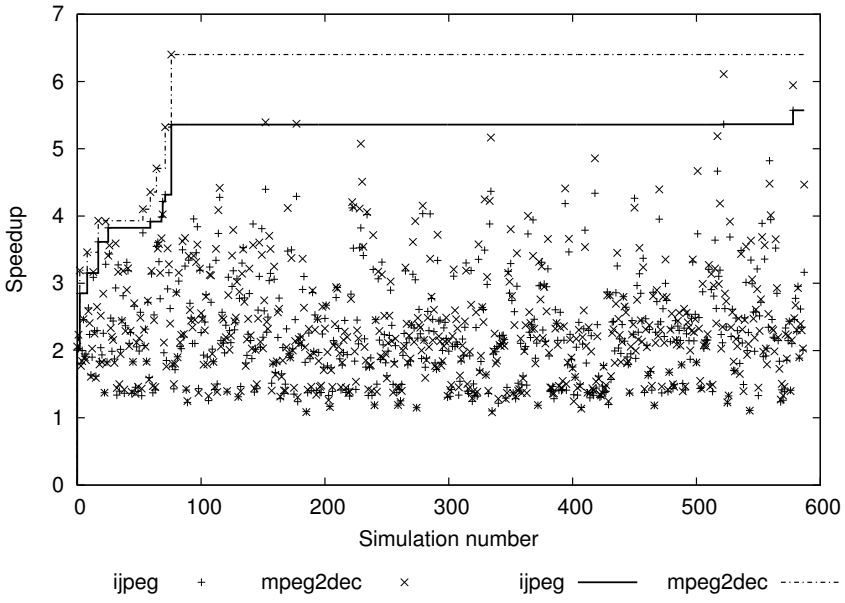


(e) area $\leq$ 17500 Mλ^2

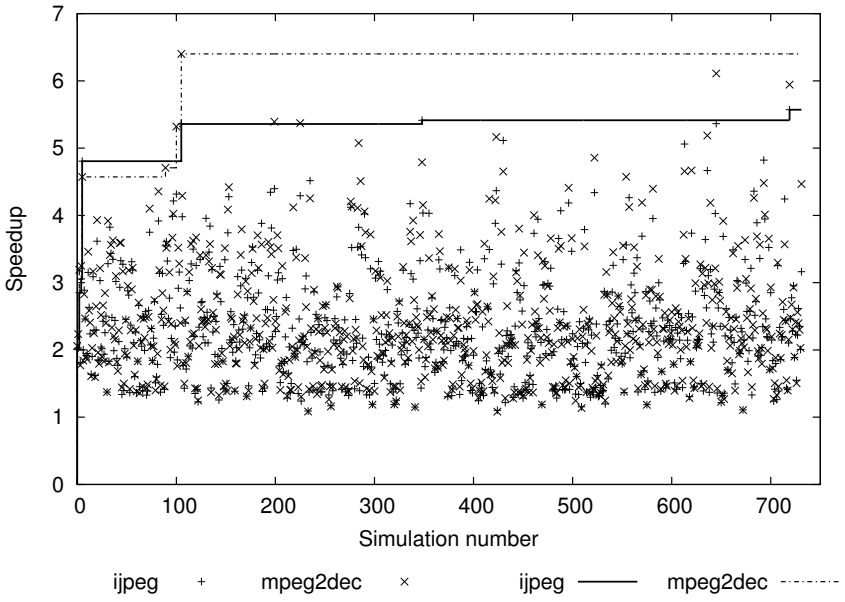


(f) area $\leq$ 20000 Mλ^2

Figure 3: Plots with various limits on area (continued)



(g) area $\leq$ 25000 Mλ^2



(h) area $\leq$ 30000 Mλ^2

Figure 3: Plots with various limits on area (continued)

minimum configuration has a speedup of 2, and so on. The upper limit of the speedup is 6.4 for the `ijpeg` simulations and 7.5 for the `mpeg2dec` simulations.

The plots are produced by iterating over the set of configurations. Once a configuration is encountered that satisfies the area restriction, it is plotted as a point in the figure. The two different lines in a figure indicate the best configuration encountered so far.

At this point, the benefits of our approach become clear. When a configuration is generated, its area is calculated first, which is a relatively cheap operation. Only if its area satisfies the area requirement, a simulation is performed. For example, when taking an area limit of 12000 Mλ^2 (Figure 3(a)), only 33 simulations have to be performed, instead of 1000, to find the best configuration in the set of all configurations. This makes a huge difference in the amount of simulation time, that is needed to find configurations that satisfy the area restriction. In fact, even after only four simulations are performed, the speedup hardly improves in this particular case. Thus, one might consider to stop the algorithm after a very small number of simulations.

In Table 3 we compare the speedup against the area of several configurations. The second, fourth and fifth best performing configurations for the `ijpeg` benchmark, are only about twice as big as the reachable minimum, but have an average speedup of 5.5. The first three and fifth best performing configurations for the `mpeg2dec` benchmark, are also about twice as big as the reachable minimum and have an average speedup of 6.0. This indicates our iterative approach is able to find a configuration that performs very well, while the area increase is relatively small.

Configuration	Speedup		Area (Mλ^2)
	`ijpeg`	`mpeg2dec`	
Reachable minimum	1.0	1.0	11250
Reachable maximum	6.4	7.5	44539
Minimal SimpleScalar configuration	0.6	0.7	11168
Huge SimpleScalar configuration	7.2	8.5	13764139
Best performing (`ijpeg`)	5.6	4.9	35166
2nd best performing (`ijpeg`)	5.6	5.9	20393
3rd best performing (`ijpeg`)	5.4	4.8	27382
4th best performing (`ijpeg`)	5.4	6.1	19491
5th best performing (`ijpeg`)	5.4	6.4	21118
Best performing (`mpeg2dec`)	5.4	6.4	21118
2nd best performing (`mpeg2dec`)	5.4	6.1	19491
3rd best performing (`mpeg2dec`)	5.6	5.9	20393
4th best performing (`mpeg2dec`)	4.8	5.6	31340
5th best performing (`mpeg2dec`)	4.4	5.4	22052

Table 3: Speedup and area of several configurations.

In Table 4, for each area limit, the best performing configuration is given. Particularly for the RUU size and the number of memory ports, a relation-

ship shows up between the parameter value and the area limit. Configurations with a small area have a small RUU size and only one memory port. Configurations with a larger area have a larger RUU size and two memory ports. For the `mpeg2dec` benchmark, a similar relation holds for the instruction cache size. However, for the `ijpeg` benchmark, the instruction cache size of the configuration with the smallest area is equal to the instruction cache size of the configuration with the largest area.

5 Discussion

In this section, we discuss the results that were presented in Section 4. Next, we evaluate the iterative algorithm applied to the search for optimal microprocessor configurations.

5.1 Results

The first thing that can be remarked from Figure 2 is that there are only a few configurations that perform very well. The majority of the configurations tend to be in the lower half of the graphs. This is caused by the fact that there exist several dependencies between the different components or tuning parameters. For example, including five integer ALU's does not increase performance that much when the instruction fetch queue size is set to one.

Another thing that can be concluded from the same figure is that the number of memory ports in a microprocessor has a huge impact on the final chip area: the effect of all other components is undone nearly entirely when the number of memory ports is increased by one. 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 two (note that we cannot substantiate this conjecture as we did not evaluate all nine million configurations).

When analyzing the performance simuluation results of each configuration, we discovered that both caches do not need to be that large for the `ijpeg` benchmark. A data cache of 2048 bytes and an instruction cache of 4096 bytes should 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 chose the right benchmark suite when designing a microprocessor.

ijpeg									
data cache	instr. cache	branch pred.	BTB	RUU	#int. ALUs	#mem. ports	FQsize	Issue width	LSQ
area $\leq$ 12000 Mλ^2									
2048	16384	512	512	16	1	1	2	2	4
area $\leq$ 13000 Mλ^2									
2048	8192	2048	512	32	5	1	4	4	16
area $\leq$ 14000 Mλ^2									
2048	8192	2048	512	32	5	1	4	4	16
area $\leq$ 15000 Mλ^2									
2048	8192	2048	512	32	5	1	4	4	16
area $\leq$ 17500 Mλ^2									
32768	16384	512	512	64	3	1	16	8	16
area $\leq$ 20000 Mλ^2									
8192	32768	4096	64	64	3	2	8	4	16
area $\leq$ 25000 Mλ^2									
8192	16384	1024	256	128	4	2	4	8	16
area $\leq$ 30000 Mλ^2									
8192	16384	1024	256	128	4	2	4	8	16
mpeg2dec									
data cache	instr. cache	branch pred.	BTB	RUU	#int ALUs	#mem. ports	FQsize	Issue width	LSQ
area $\leq$ 12000 Mλ^2									
1024	2048	512	512	8	5	1	2	2	16
area $\leq$ 13000 Mλ^2									
2048	8192	2048	512	32	5	1	4	4	16
area $\leq$ 14000 Mλ^2									
2048	32768	1024	128	64	3	1	4	4	8
area $\leq$ 15000 Mλ^2									
2048	32768	1024	128	64	3	1	4	4	8
area $\leq$ 17500 Mλ^2									
32768	32768	2048	512	32	4	1	16	4	16
area $\leq$ 20000 Mλ^2									
8192	32768	4096	64	64	3	2	8	4	16
area $\leq$ 25000 Mλ^2									
2048	32768	2048	256	128	5	2	8	4	16
area $\leq$ 30000 Mλ^2									
2048	32768	2048	256	128	5	2	8	4	16

Table 4: Configurations found for each area limit

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 `ijpeg` 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.

5.2 Algorithm Evaluation

As can be concluded from the previous two paragraphs, one should carefully consider with which benchmark the performance simulation step has to be done. The chosen benchmark should reflect the application of the final product closely, or else one ends up with a microprocessor that is optimized for the wrong purpose.

Figure 3(a) shows that only 33 instead of one thousand simulations are examined, since the remaining configurations exceed the area limit. Because we examine a fixed set of 1000 different configurations, only a relatively small number of configurations is returned by our approach. This is caused by the fact that, when the area restriction is tight, the number of configurations that satisfy the restriction is small. However, because of the enormous reduction in simulation time needed, one can evaluate many more configurations, since the area calculation step is a very fast operation compared to the simulation step.

Furthermore, while the area limit of 12000 Mλ^2 is only slightly more than the area of the reachable minimum configuration, we still gain a speedup of a factor of about 2.5. This shows that carefully selecting a few additional resources can be highly effective. When the area limit is between 13000 and 15000 Mλ^2, speedups of about 4 are already achieved. When a larger area is allowed, the speedup is about 1.0 below the reachable maximum for each benchmark. This stresses the effectiveness of our approach.

The plots in Figure 3 show that after about 50 iterations a configuration has been found that satisfies the area restriction and performs already significantly better than the minimum configuration. In general, after about 200 iterations a configuration has been found that will scarcely be exceeded in performance by a configuration from the set of remaining configurations.

However, an important limitation of our approach is still time. Performing a single performance simulation requires a lot of computation time. Especially when one needs a reliable result, the simulation should involve enough cycles to

correctly represent the intended purpose of the final product. Therefore, finding configurations with the algorithm we proposed in Section 3.1 still remains quite a time-consuming task.

6 Future Work

In this section we discuss some future work that deals with our proposed approach of finding suitable microprocessor configurations.

In the previous sections, we have evaluated the most basic implementation of the iterative approach, namely, the random generation of parameter values. The results of the performance simulations are not used for any feedback in the algorithm. Of course, most of the enhancements that are invented in the domain of iterative compilation can be applied to our approach as well. We highlight one particular alternative: the use of a genetic algorithm.

The algorithm starts with a small population of individuals with random parameter values. The quality ("fitness") of an individual configuration depends on two statistics: the performance of the configuration and an estimate of the chip area. Therefore, the $performance : area$ ratio may provide a suitable fitness function. Several genetic operators (like mutation and crossover) can be applied to the individuals of the population, as these are just a set of integer values. After a number of iterations the algorithm should stop. The population will hopefully consist of configurations that perform better than the initial set of parameter values.

Another possible direction in which this research can be extended is by applying data mining techniques on the obtained data, which consists of the configurations together with their estimated area and computed performance. One might discover interesting patterns and dependencies between the different tuning parameters mutually, and between the parameters and the area. The results of this kind of analysis can be used 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, as this turns out to be a very compute intensive and therefore time consuming step. When searching for a processor configuration that has to serve a specific application, this would not be that hard. But for an all-purpose microprocessor - like the one of a cell phone, which nowadays has to be able to do image and video processing, sound recording and playing games, besides handling a phone call - the performance simulation step might pose a serious restriction on the number of iterations the algorithm could perform in a reasonable amount of time. A possible way to improve this, is to use small, but representative inputs for the benchmark applications used in the simulations. In the next section we discuss some alternative solutions for the simulation step.

7 Related Work

Design space exploration has been studied extensively in order to come up with more efficient microprocessors and to improve the design process. Our approach is in fact based on the Y-chart approach, which was proposed by Kienhuis et al. in [11] and further investigated in [12]. In [11], the authors already indicate that the simulation phase is the bottleneck of the approach: the more efficient this phase is, the more useful the Y-chart approach will be (since more candidate designs can be evaluated in reasonable time). Our results clearly agree with this observation.

Various solutions have been proposed in order to make the performance analysis step faster. The Artemis project [13, 14] is meant to support the design of embedded systems. Two different frameworks for the simulation phase are adopted in this project: Spade [15], which provides a model for (rapid) high level architecture performance simulations, and Sesame [16], which provides a method for evaluating designs at multiple abstraction levels. With the latter, one can specify some of the architecture components at a high level and specify other components at a lower and more detailed level. In [17, 18], the Sesame method is investigated more thoroughly; in [19], the mechanisms for calibration of the Sesame framework are discussed.

Another alternative for the simulation step is to make use of statistical simulations. In [20], several statistical simulation models are evaluated, with different degrees of detail. Simple statistical models yield an execution time of only a few seconds. This could be very useful in fast (but less accurate) design space exploration. In [21], an hybrid analytical-statistical model is presented in order to support early design stage architecture evaluations. In [22], the statistical simulation approach is investigated more thoroughly, by taking power consumption and cycle time also into account.

In [23], an analytical approach for determining the parameter configuration of a cache in an embedded system is discussed. The authors propose a method that takes a design constraint (e.g., desired number of cache misses) and a trace-file as input, and outputs a set of optimal cache configuration parameters (in [23] they limit this set to only two parameters: cache size and degree of associativity; other parameters are set to a fixed value). This method appears to be much faster than the traditional exhaustive performance simulations. However, it is a very complex task to develop an analytical model for each component. Developing a model for our entire parameter set might be very hard to realize, because of the amount of different parameters and the various dependencies between the parameters.

Some effort has been spent on minimizing the parts of the design space that have to be evaluated. In [24], Haubelt and Teich propose a method for the synthesis of system configurations using Pareto-Front Arithmetics. This approach leads to a dramatically reduced exploration time. In [25], a framework called Model based Integrated simuLAtioN (MILAN) is extended with two tools (DESERT and HiPerE) in order to speedup the exploration process. These allow (among other features) for rapid estimation and optimization of energy

consumption.

Instead of focussing primarily on alternative architecture instances, it is also possible to evaluate different target application instances, as described by Stefanov et al. in [26]. With their method, the target application is decomposed several times in several different concurrent tasks (with a different degree of exploited task-level parallelism). As stated in [26], the performance of an architecture can significantly depend on the application instance that is used.

In the same domain of taking the target application into account when desiging an embedded system, lies the *Compaan*-project [27, 28, 29]. Compaan takes a sequential imperative Matlab program as input and compiles it into a process network. This concurrent representation provides a better foundation for exploiting parallelism. According to Rijpkema et al. in [27]:

> Novel high-performance, domain specific, embedded architectures are often composed of a microprocessor, some memory, and a number of dedicated coprocessors.

These kind of architectures are to be used in video consumer appliances, adaptive radar processing and mobile communication devices, as indicated by Kienhuis et al. in [28]. The Compaan toolset tries to assist designers during the process of partitioning the target application into hardware and software implementations.

In [30], the authors investigate an extension to Compaan, called *Laura*. This tool takes a process network specification generated by Compaan as input and transforms it into a design implementation described as synthesizable VHDL. With this VHDL code the original Matlab program (as provided to Compaan) can be programmed onto a Field Programmable Gate Array (FPGA) platform. Stefanov et al. present a system design approach based on the Compaan/Laura tool chain in [31].

An interesting and completely different proposal is made by Pimentel in [32]. The author pleads for the development of methods that provide assistance by visualization during the design space exploration process. Pimentel states that visualization is an important tool in other domains and thus could be useful in the domain of computer architecture too. However, no concrete methods are proposed in his paper.

The structure of the approaches described by Eyerman et al. in [33] is very close to our method. The authors use several search algorithms, including a random-based algorithm and a genetic search algorithm (we briefly discussed a proposal of the latter in the previous section). They also use `sim-outorder` from the SimpleScalar toolset for the performance simulation step, together with a tool (called *Wattch*) that estimates the energy consumption during the simulation. Furthermore they combine the SimpleScalar simulation with a statistical simulation into a two-phase simulation, in order to prune the design space.

However, most of the methods and approaches discussed in this section do not take the physical chip area into account (except for [30], where the authors compare the estimated area of the output of their tool with the area of existing

IP cores). In general, researchers tend to focus primarily on the performance (and in some cases also the energy consumption) of the proposed configuration.

8 Conclusion

In this paper we have shown that an iterative approach to the problem of finding suitable microprocessor configurations works reasonably well. It finds a suitable configuration (that satisfies a given area restriction) after a relatively small number of iterations. Furthermore, we have shown that even a small increase in the resources compared to a minimal configuration can give a speedup of about 2.5, which implies that tuning a processor with our approach can be highly effective. However, we also found that the approach is quite compute intensive. Nevertheless, this approach could be quite useful in the field of design space exploration.

References

[1] M. Levy, Multithreaded Technologies Disclosed at MPF, Microprocessor Report, November 2003.

[2] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 1996

[3] T. Kisuki, P.M.W. Knijnenburg and M.F.P. O'Boyle, Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation, Proc. Parallel Architectures and Compilation Techniques PACT2000, pages 237-246, 2000.

[4] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle and H.A.G. Wijshoff, Iterative Compilation in Program Optimization, Proc. 8th Int'l Workshop on Compilers for Parallel Computers, CPC2000, pp. 35-44, 2000.

[5] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg, Evaluating Iterative Compilation, In Proc. Languages and Compilers for Parallel Computers (LCPC02), pp. 305-315, 2002.

[6] M. Steinhaus, R. Kolla, J. Larriba-Pey, T. Ungerer and M. Valero, Transistor Count and Chip-Space Estimation of SimpleScalar-based Microprocessor Models, In Proc. of the Workshop on Complexity-Effective Design, June 2001. 12.

[7] SimpleScalar LLC, SimpleScalar Tools, available from: `http://www.simplescalar.com/tools.htm`, last accessed: May 31st, 2006.

[8] Standard Performance Evaluation Corporation, SPEC CINT95 Benchmarks, available from: `http://www.spec.org/cpu95/CINT95/index.html`, last accessed: May 31st, 2006.

[9] MPEG Software Simulation Group, MPEG-2 Video Codec, available from: `http://www.mpeg.org/MSSG/`, last accessed: June 1st, 2006.

[10] S. McFarling, Combining branch predictors, Digital Western Research Lab. Technical Report #TN-36, June 1993.

[11] B. Kienhuis, E. Deprettere, K. Vissers and P. van der Wolf, An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures, In Proc.

11-th Int. Conf. on Application-specific Systems, Architectures and Processors, 1997.

[12] B. Kienhuis, E. Deprettere, P. van der Wolf and K. Vissers, A Methodology to Design Programmable Embedded Systems, In the LNCS series of Springer-Verlag (c), Volume 2268, SAMOS: Systems, Architectures, Modeling, and Simulation, editors Ed F. Deprettere, Jurgen Teich, and Stamatis Vassiliadis, November 2001

[13] A. D. Pimentel, P. Lieverse, P. van der Wolf, L.O. Hertzberger and E.F. Deprettere, Exploring Embedded-Systems Architectures with Artemis, In IEEE Computer, pp. 57-63, Vol. 34 (No. 11), Nov. 2001.

[14] A. D. Pimentel, P. van der Wolf, E. F. Deprettere, L.O. Hertzberger, J. T. J. van Eijndhoven and S. Vassiliadis, The Artemis Architecture Workbench, In the Proc. of the Progress workshop on Embedded Systems, pp. 53-62, October 2000.

[15] P. Lieverse, P. van der Wolf, E. Deprettere and K. Vissers, A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems, In Proceedings 1999 Workshop on Signal Processing Systems (SiPS'99), pp. 181-190, 1999.

[16] A. W. van Halderen, S. Polstra, A. D. Pimentel, and L.O. Hertzberger, Sesame: Simulation of embedded system architectures for multi-level exploration, In Proc. of the conference of the Advanced School for Computing and Imaging (ASCI), pages 99–106, May 2001.

[17] A. D. Pimentel, S. Polstra, F. Terpstra, A.W. van Halderen, J. E. Coffland and L.O. Hertzberger, Towards Efficient Design Space Exploration of Heterogeneous Embedded Media Systems, In E. Deprettere, J. Teich and S. Vassiliadis (eds), Embedded Processor Design Challenges: Systems, Architectures, MOdeling, and Simulation (SAMOS), LNCS, number 2268, pp. 57-73, 2002

[18] J. E. Coffland and A. D. Pimentel, A Software Framework for Efficient System-level Performance Evaluation of Embedded Systems, In the Proc. of the 18th ACM Symposium on Applied Computing (SAC), Embedded Systems track, pp. 666-671, Melbourne, Florida, USA, March 2003.

[19] A. D. Pimentel, M. Thompson, S. Polstra and C. Erbas, On the Calibration of Abstract Performance Models for System-level Design Space Exploration, to appear in the Int. Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (IC-SAMOS 2006), IEEE CAS.

[20] S. Nussbaum and J. E. Smith, Modeling Superscalar Processors via Statistical Simulation, In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001), pages 15-24, September 2001.

[21] L. Eeckhout and K. De Bosschere, Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces, In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001), pages 25-34, September 2001.

[22] L. Eeckhout, D. Stroobandt, and K. De Bosschere, Efficient Microprocessor Design Space Exploration through Statistical Simulation, In Proceedings of the 36th Annual Symposium on Simulation, March 2003.

[23] A. Ghosh, T. Givargis, Analytical Design Space Exploration of Caches for Embedded Systems, In Design, Automation and Test in Europe Conference and Exhibition (DATE'03), pages 650-655, March 2003.

[24] C. Haubelt, J. Teich, Accelerating Design Space Exploration Using Pareto-Front Arithmetics, In Proceedings of Asia and South Pacific Design, Automation and Conference, pages 525-531, 2003.

[25] S. Mohanty, V. K. Prasanna, S. Neema, J. Davis, Rapid Design Space Exploration of Heterogeneous Embedded Systems Using Symbolic Search and Multi-Granular Simulation, In Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems LCTES/SCOPES '02, Volume 37 Issue 7

[26] T. Stefanov, B. Kienhuis and E. Deprettere, Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances, In Proc. 10th Int. Symposium on Hardware/Software Codesign (CODES'02), pp. 7-12, May 2002.

[27] E. Rijpkema, B. Kienhuis, and E. Deprettere, Compilation from Matlab to process networks, In Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES), pages 388-395, October 1999.

[28] B. Kienhuis, E. Rijpkema, and E. Deprettere, Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures, Presented at the 8th International Workshop on Hardware/Software Codesign (CODES'00), May 2000.

[29] Leiden Institute of Advanced Computer Science, Compaan, available from: `http://www.liacs.nl/~cserc/compaan/`, last accessed: July 11th, 2006.

[30] C. Zissulescu, T. Stefanov, B. Kienhuis, E. Deprettere, Laura: Leiden Architecture Research and Exploration Tool, Presented at the International Conference on Field Programmable Logic and Applications (FPL), September 2003.

[31] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, E. Deprettere, System Design using Kahn Process Networks: The Compaan/Laura Approach, In Proceedings of the Design, Automation and Test in Europe conference DATE2004, February 2004.

[32] A. D. Pimentel, A Case for Visualization-integrated System-level Design Space Exploration, In Proceedings of the 5th Int. Workshop on Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS 2005), pp. 455-464, LNCS, July 2005.

[33] S. Eyerman, L. Eeckhout, K. De Bosschere, Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors, In Design and Test in Europe (DATE), March 2006.