TriMedia CPU64 Design Space Exploration

G.J. Hekstra, G.D. La Hei, P. Bingley, F.W. Sijstermans*
Philips Research Labs, Eindhoven, The Netherlands
**Philips Semiconductors TriMedia, Sunnyvale CA, USA*
hekstrag@natlab.research.philips.com

Abstract

Within Philips Research Labs, we are investigating the 64-bit VLIW core for future TriMedia processors. We have performed an extensive Design Space Exploration (DSE) on this core using quantitative analysis, using a benchmark suite of applications which are representative for multimedia processing. We have explored, among others, the configurations of the different functional units (FUs) of the 64-bit VLIW core. We show the merit of our approach and of the environment that we have developed for design space exploration.

1. Introduction

The design space exploration of the TriMedia 64-bit core (CPU64) [1] is carried out at Philips Research Labs, The Netherlands, in close cooperation with Philips TriMedia in Sunnyvale CA, USA. The aim is to develop a hardware and software platform to support the development and real-time execution of demanding multimedia applications. It is not the intention to start these developments from scratch. The current TriMedia processor, the TM1x00, is our starting point.

1.1. TM1x00 architecture

The TM1x00 architecture, as shown in Figure 1, comprises a Very Long Instruction Word CPU core with caches, a global bus (called data highway), and a collection of dedicated co-processors. Most of the co-processors perform I/O tasks, whereas a few others perform dedicated processing tasks, *e.g.*, variable length decoding for MPEG2 video. The major part of the data processing is done in software on the VLIW core.

Both the co-processors and the instruction set of the VLIW core are geared to the domain of media processing, which comprises the following application areas:

- Data communication
- Audio/video processing and coding
- Graphics

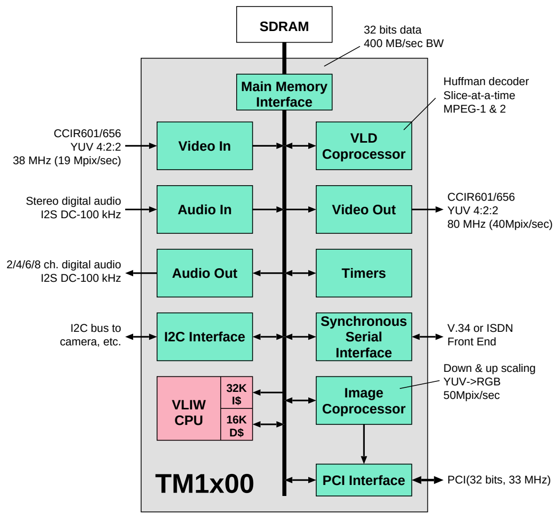

Figure 1. The TM1x00 (TM1000 and TM1100) architecture

The performance of a VLIW core strongly depends on the extent of exploitation of the parallel VLIW architecture. We can identify parallelism at two different levels in the TriMedia core. First, being a VLIW machine, the core is equipped with five issue slots. This means that a single instruction contains up to five operations, which are issued in parallel. Second, the TriMedia core features a set of Single Instruction Multiple Data (SIMD) operations or vector operations. The operands of such operations are split up into separate elements (four bytes or two 16-bits operands in the case of TM1000). In video processing, for instance, each byte can contain a pixel value.

1.2. Development of CPU64

To cope with the high demands of near future media applications, such as digital high definition television in the

USA, a significant performance increase must be achieved (estimated at a factor of six) by the successors of the TM1x00 processor. The overall performance of the TriMedia architecture is determined by several key-factors, such as: bandwidth of the data highway, well chosen dedicated co-processors to support important applications, size and type of the caches, and last but not least the performance of the VLIW core.

The main focus has been on the design of a high performance 64-bit VLIW core which will replace the current 32-bit TriMedia core. The CPU64 aims at achieving the required performance by doubling the data word size (to 64-bits), higher clock frequency, improved cache design, and not in the least by a powerful vector instruction set [1][2][3].

The development of CPU64 is a complicated task with many different aspects. It not only covers architecture and instruction set design but also tooling, which is considered as being important. Retargetable compilation and simulation tools were developed to enable the mapping of applications on CPU64 to verify and evaluate its performance.

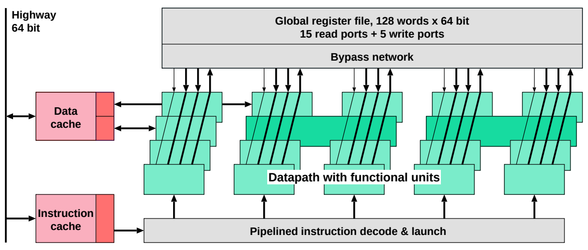
Highway
64 bit
Global register file, 128 words x 64 bit
15 read ports + 5 write ports
Bypass network
Data cache
Datapath with functional units
Instruction cache
Pipelined instruction decode & launch

Figure 2. Diagram of the CPU64 VLIW core

Figure 2 shows the architecture of the CPU64. On the left hand side we see the data and instruction caches which are connected to the 64-bit data highway. The core has access to background memory via this path. The functional units are distributed over the five issue slots of the core. A single VLIW instruction contains the operations for each issue slot. In a single issue slot, a functional unit has the availability of two 64-bit read ports and a single 64-bit write port to the register file. An additional single bit read port is present for guarded execution. Via the bypass network the result of an operation can be used directly as an operand to another operation without intermediate storage in a register. The CPU64 also features functional units that lay across two adjacent issue slots. So-called super operations (super-ops) are executed on these units. A super-op can have up to four 64-bit operands and up to two 64-bits results.

There are several different types of functional units (FUs). Each type of FU supports a specific set of operations. In all, there are FUs to perform byte shuffles, bit shifts, multiplications, integer and floating point arithmetic, look-up table operations, load/store operations, branches, and special operations (cache, MMU, MMIO).

The new instruction set, provided by these FUs, contains many powerful SIMD vector instructions. Figure 3 shows an example of a vector operation. This instruction, `ub_avg`, calculates the element-wise average of two 64-bit vectors. In the example each vector consists of eight unsigned bytes.

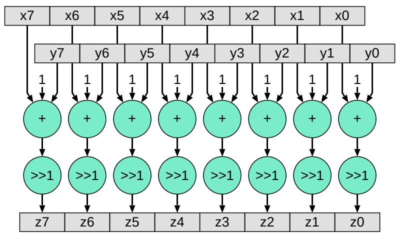

Figure 3. Example of a CPU64 vector operation: `ub_avg`

1.3. Problem definition

In the design of such an advanced processor many decisions must be taken. The trade-off between performance and silicon-area plays a crucial role in the design process. There are many machine parameters that influence this trade-off. For example, looking at the caches, we can vary the size, number of ports, associativity and replacement strategy. The task that we consider to be the most challenging is that of studying the impact of the quantity and placement of the functional units.

This leads to the following objective:

> *We want to systematically explore the design space of the CPU64, in order to provide quantitative data on which design decisions can be based.*

Related to this objective we identify the following subproblems:

- We need to obtain a number of representative multimedia applications. Together they form a benchmark by which the performance of the CPU64 can be measured.
- We require an integrated framework in which we can vary design parameters, run experiments, and observe the results.
- We need to conceive experiments that enable us to extract the knowledge on which design decisions can be based. These experiments must be well designed (both reliable and feasible).

Clearly, the whole of the problems stated above is of a complex and multi-disciplinary nature. It demands a systematic and structured approach, which is the subject of the next section.

2. DSE Approach

In [4] Kienhuis proposes a general scheme for DSE which he refers to as the Y-chart. This scheme is shown below in Figure 4. The Y-chart scheme reflects the essence of design space explorations, including that for CPU64. The backbone of the scheme consists of a *retargetable* compiler and simulator. The input to the compiler is both an *architecture description*, and a number of *benchmark applications*. The output at the simulator are numbers that reflect the system performance. Note that the system performance depends on the quality of *all* above mentioned components.

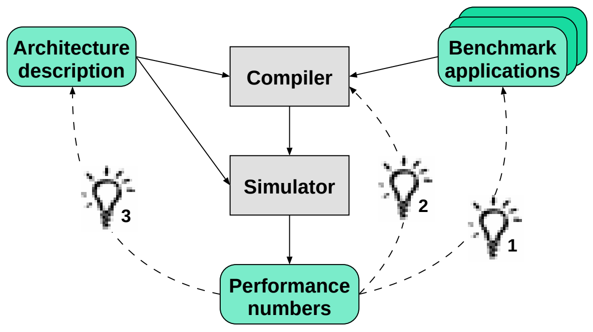

Figure 4. The Y-chart approach

The scheme shows three possible iteration loops that affect this system performance:

1. **Application code**
   Developers iterate this loop in the process of optimizing their application code.
2. **Toolchain**
   This loop is iterated by toolchain developers to enhance the performance of the compilers.
3. **Architecture**
   This loop is iterated in the exploration of the design space of the architecture.

In this paper, we focus on the third loop. We require that the quality of both the toolchain and the application code are such that we can reliably measure the effects of variations in the architecture.

We also require an environment in which we can conduct experiments that measure the performance of different variations in the architecture.

The aspects relating to the multimedia benchmark are dealt with in Section 3 of this paper. The integrated DSE environment, specific for CPU64, is described in Section 4. The design of experiments, applied to the problem of exploring the functional unit configuration, is described in Section 5.

3. Multimedia Benchmark

In order to obtain realistic and reliable results from the experiments in the DSE, it is required to impose a representative load on the CPU64. This load should be comparable to the average load imposed by applications that are part of the application domain of the CPU64 [5]. The benchmarks should process representative input data and they should be optimized such that they run efficiently and use the special vector operations as offered by the CPU64 core [6].

Due to these demands, we face the following challenges:

- Which benchmark applications do we use?
- How to make all this software manageable?
- How should we weigh the individual loads such that a representative load is imposed on the CPU64?

The applications included in the current benchmark suite are listed in Table 1.

Table 1. The benchmark suite

Category	Application
Data communication	Viterbi decoding
Audio coding	AC3 decode
Video coding	MPEG2 encode
	DVC decode
Video processing	Layered natural motion
	Dynamic noise reduction
	Peaking
Graphics	3D renderer back-end
	Mesa 3D graphics library

To ensure that we can maintain and use all benchmark software, which originates from different developers, we have provided them with our Application programmer's ToolKit (ATK). This toolkit enforces a standardized directory and Makefile structure upon the program. This approach enables easy integration of the individual applications into a benchmark suite for use in the DSE environment.

All the individual benchmark applications impose a certain load on the CPU64. To obtain a figure which reflects the overall performance of the CPU64 core these individual loads should be combined.

We apply the following approach. Imagine a near future high-definition digital TV-set which features a CPU64. For this TV-set we have defined different modes of operation. In each mode of operation several applications are required to run concurrently. Averaging over all the modes, we obtain a figure how often an application occurs. Given a certain CPU64 configuration, we can simulate, for each application, how many cycles it would consume in one second of real-time operation. An overall performance figure for this configuration is then obtained by weighing the consumed cycles per application by how often they occur.

3.1. Characteristics of the benchmark suite

It is the intention of the benchmark suite to cover the application domain of the CPU64. Since this domain is rather broad, the applications in the benchmark exhibit different characteristics. Some of the applications are more complex than others, and some are more memory-I/O intensive than others.

A characteristic of the benchmark, which is important w.r.t. the DSE, is the utilization of the functional units. In Figure 5 the overall utilization per functional unit is shown. Note that the vertical axis is logarithmic. The benchmarks were weighted as described in the previous section to obtain these figures.

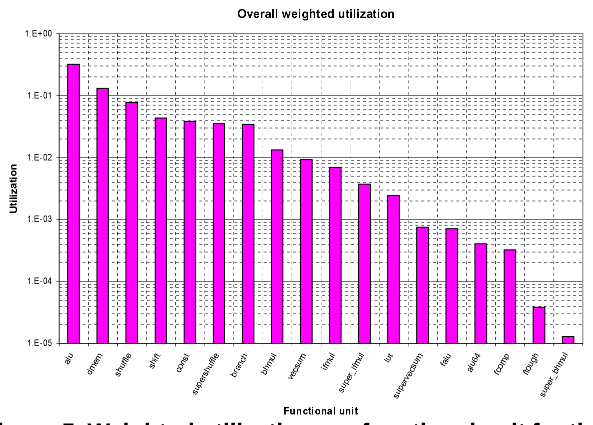

Figure 5. Weighted utilization per functional unit for the whole benchmark suite

Analysis of Figure 5 leads to the following observations:

- Only a few FUs (`alu`, `dmem`, `shuffle`) are used intensively. This property is exploited in the experiment set-up later on.
- Some FUs are not used by the benchmarks. The quantity and placement of these units can not be determined by our DSE. For these units, the quantity and their placement depends on other criteria (such as design experience, floorplan optimization, or propagation delay issues).

4. DSE Environment

The environment in which we perform our DSE experiments is based on the `Nelsis` framework [7][8]. In this framework a design flow can be defined. `Nelsis` has been used before in other DSEs, such as that of a super-scalar MIPS architecture. To be able to use `Nelsis` as an environment for exploring CPU64, some additional and specific tools were required. These include visualization tools to interpret results, simulation tools and glue software for system integration.

`Nelsis` automatically keeps track of the relations between generated output data and the tools and application code used to obtain these results. This ensures that our experiments are reproducible, which is essential.

The DSE can be controlled from a sophisticated graphical user interface (see Figure 6) which visualizes the design flow. Note that each different exploration has its own unique design flow, which has to be defined prior to the actual DSE.

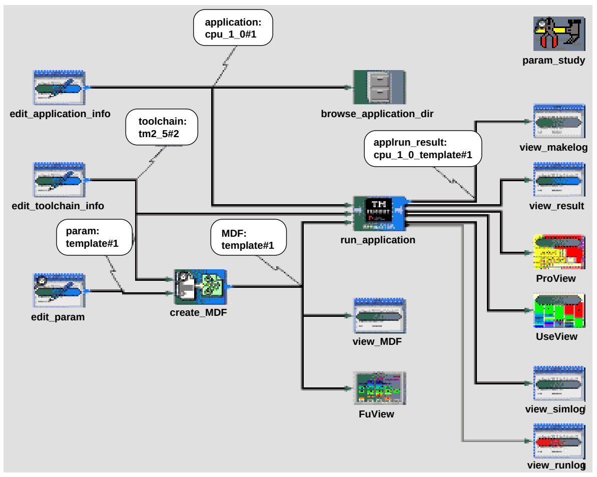

Figure 6. The CPU64 DSE design flow as displayed by the `Nelsis` graphical user interface

On the left hand side in Figure 6 one can see the inputs to an experiment. There are three different inputs: *application info*, *toolchain info* and the *parameters*. In the application info file the benchmark applications which take part in the experiment are defined. In the toolchain info file one of the toolchain releases is selected. With the aid of the parameters a functional unit configuration is defined for the machine under test.

Before the simulation can start a *machine description file* (MDF) must be generated. An MDF defines, among other things, the configuration of functional units of the CPU64 core and it forms the target for the retargetable simulation tools [6]. The MDF is derived from the parameters in combination with a template MDF. The block ‘create_MDF’ in Figure 6 represents this functionality.

The compilation and simulation process is represented by ‘run_application’ in Figure 6.

We have developed *pseudo-simulation*, as a fast and accurate alternative to lengthy machine simulation. `Pseudosim` delivers *cycle accurate* results for instruction cycles, which works well for VLIW machines, where the schedule is determined at compile time. A single machine simulation is performed to obtain statistics such as execution counts and branch probabilities. Based on these statistics, the simulation results for an arbitrary machine can be calculated by `pseudosim` in a much shorter time. The normal machine simulation time of 18 hours for our multimedia benchmark is reduced to around 4 *minutes*.

After running the experiment the results can be examined either textually or in a more visually attractive manner using some of our dedicated CPU64 DSE tools. Note that `Nelsis` allows the user to run experiments both interactively or as a batch job.

5. Exploration of the Design Space

In essence, the design space exploration of the CPU64 covers the whole range of possible machine parameters. Not all of these need to be explored, as they may be fixed by external requirements beforehand.

We focus in this paper only on the problem of functional unit configuration: to determine how many of each type of functional unit are needed, and where to place these in the issue slots. This is a problem that leads to a literally immense design space and has, by far, been the most challenging.

The inherent difficulty of functional unit configuration is the following. The configuration of a certain type of FU cannot, in general, be seen independent from the choice of placement of the other FUs. This almost naturally implies that the whole design space must be searched exhaustively to find the optimum placement.

This problem is the topic of the remainder of this section, where we start out to investigate how large the design space actually is, and what fraction of it we really need to explore.

5.1. Analysis of the size of the design space

The CPU64 core has 5 issue slots, in which 30 different types of FUs are placed. Of these types of FU, 24 occupy a single issue slot while the remaining 6 are super-op FUs that occupy 2 adjacent issue slots.

Let us consider the following, naive calculation of the size of the design space at hand. There are

$$\binom{5}{1} + \binom{5}{2} + \binom{5}{3} + \binom{5}{4} + \binom{5}{5} = 31$$

possible configurations for an FU that occupies a single issue slot. The number of configurations for super-op FUs is much less, being 7. The size of the design space for a full exhaustive exploration then amounts to:

$$S = 31^{24} \times 7^{6} \approx 10^{41}$$

Clearly, this design space is far too large to explore exhaustively. Note that we disregard the effect of permutations of the issue slots, which implies that the above figure contains multiple, equivalent machines. Even then, this factor is at most 5! = 120, and only if all of the issue slots of a certain machine have a different filling of FUs. In practice, this factor is much lower, due to the restrictions of super-op FUs on the permutations. In any case, it does not make exhaustive exploration feasible.

If we take more detailed design knowledge at hand, we can further reduce the size of the design space. The naive calculation did not take into account the fact that dependencies exist in the placement of certain FUs. It also assumed that the amount of each type of FU was varied between the absolute minimum and maximum, being 1-5 for single slot FUs, and 1-2 for super-op FUs. This may not be sensible from a design standpoint. If we *do* take these factors into account, grouping dependent FUs, and placing realistic limits on their amount, we end up with a design space of a smaller size that amounts to:

$$S' \approx 10^{15}$$

However, we still consider this to be far too large for an exhaustive search, as we shall show later. Note that the restrictions that we have imposed on the design space in this calculation were based on (previous) design knowledge, and not on actual measurements.

Clearly we need to develop a strategy that avoids a full exhaustive search of the design space. We also want to base decisions on quantifiable data, rather than purely on design knowledge. We tackle this problem in the following way. First, we probe the design space, to determine its characteristics, see Section 5.2. We use the obtained knowledge to restrict and structure the search in the design space. Second, we systematically explore the design space in only those regions of interest as indicated by the initial probing, see Section 5.3.

Experiment set up is of importance. We need to conceive experiments that enable us to extract the knowledge on which design decisions can be based. These experiments must be well designed, which means that they are both reliable and feasible. Feasible in the sense that the design points can be explored in reasonable time.

5.2. Probing the design space

In the initial probing of the design space, we ask ourselves the following questions:

- What are realistic bounds for the amount of certain types of FUs?
- Does the *whole* set of parameters need to be explored exhaustively?

To answer the first question, we have set up a number of Taguchi experiments where we vary only a single parameter at a time (varying the amount of a single FU type). The configuration of the other FUs is such that they have the least possible influence on the outcome of the experiment. The result of these experiments is a set of lower bounds for every FU type. These bounds help us to restrict the design space.

Concerning the second question, we state the following. For those parameters whose combined effect cannot be ignored we have to perform an exhaustive search. These parameters are strongly non-orthogonal. Those parameters whose combined influence is almost negligible can be left out of this exhaustive search, and treated separately.

We have set up and evaluated an experiment that measures the effect when different FUs overlap in the same issue slots. This has led to an ordering of FUs according to their influence. This ordering closely follows that of the distribution of operation issues over the FUs, as shown in Figure 5.

If we look at Figure 5 in more detail, we see that a small fraction, being 30% of the FU types, is responsible for the larger part of the utilization, being 93% of the issued operations. The other 70% of the FU types is responsible for the remaining 7% of the issued operations. This is also illustrated in Figure 7.

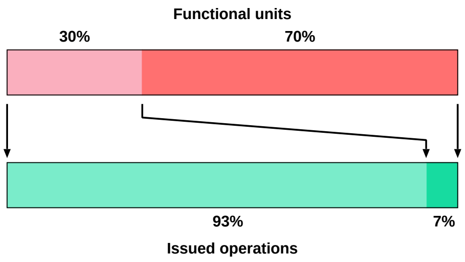

Figure 7. Distribution of issues over functional units

We use exactly the division of Figure 7, and define the set of parameters that determine the configuration of these FUs in the 30% category. These have to be combined in a single exhaustive search. The set of parameters that determine the configuration of FUs in the 70% category are explored individually. This systematic exploration of the design space is the subject of the next section.

Note that the location of this division is, in some sense, arbitrary. We have chosen it such that the estimated amount of design points (simulated machines) can be explored in reasonable time.

5.3. Systematic partitioning and exploration of the design space

In the previous section we have introduced a partitioning of the design space. The parameters that determine the configuration of the FUs in the 30% category are combined in a single, exhaustive experiment. We apply the restrictions on the amount of FUs to further limit the design space spanned by these parameters. Additionally, we exclude all permutations that result in equivalent machines.We keep the other parameters at such values, that they have minimum influence on the outcome. The result of this action is that roughly 3000 essentially different machines remain.

Figure 8 shows the area and performance characteristics of these machines. The relative area is shown along the x-axis, while the instruction cycles (weighed over the benchmark, see Section 3) are shown along the y-axis.

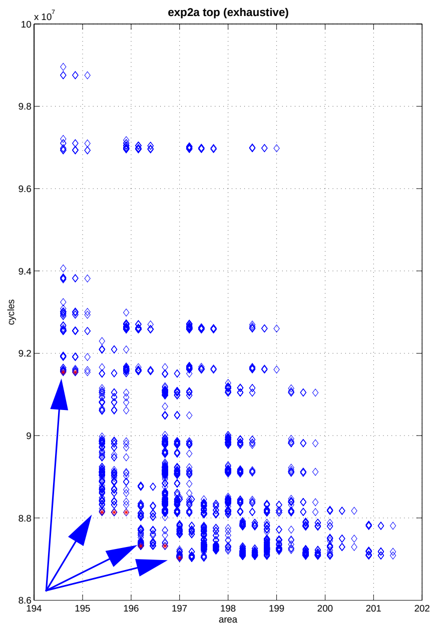

Figure 8. Outcome of the exhaustive experiment

We see that there are four machines which are superior: there are no machines that have a better performance *and* have a smaller area. These are located at the protruding bottom-left corners as indicated by the arrows in the graph.

These four machines form the starting population of the next set of experiments. In the design space exploration, we are not looking for a single global optimum, but rather for a range of machines along the area/performance trade-off curve.

The remainder of the parameters, which correspond to the FUs in the 70% category, are fixed in a series of greedy experiments. The sequence of experiments is ordered in decreasing amount of issues, as shown in Figure 5.

Per experiment we set up a starting population, which is in essence the surviving, fittest machines of the previous experiment.

For each member of this population, we produce the offspring by varying the parameter in question, and simulating the resulting machines. Next, we select the superior machines from the offspring of the entire population. These machines form the population of the next experiment.

As the greedy experiments develop, the number of superior machines along the area/performance curve increases. In order to keep each greedy experiment feasible we, in some occasions, prune the population by removing machines with poor performance or high area costs.

After the last greedy experiment, we obtain a range of machines, each with different area/performance characteristics. They are potential candidates for the final CPU64 design, and form a reference frame for further design space exploration, as we shall see in the next section.

6. Evaluation

In this section, we look at some aspects that shows the merit of our specific DSE approach.

In Table 2 we show how many design points were explored for the different stages of the experiments.

Table 2. Number of simulated machines

experiment	name	no. of machines
initial probing	exp1	185
exhaustive	exp2a	3023
greedy	exp2b	2519
total		5727

We see that the exhaustive and greedy parts of the second set of experiments are roughly of the same order. Whereas for the exhaustive part the number of design points was fixed beforehand, the number for the greedy part was dependent largely on how the population of superior machines developed. The number of design points for the initial probing of the design space is only a fraction (3%) of the total. All in all, the total ends up at a little under 6000 simulated machines.

If we take these 6000 design points as a reference, then in Table 3 we show how much time is needed to evaluate them, given different approaches.

Table 3. Simulation times

simulated machines	10^{15}	6000
machine simulation[a]	2×10^{12} years	$12\frac{1}{2}$ years
pseudo simulation[a]	7.6×10^{9} years	400 hours
machine execution[b]	3.47×10^{7} years	107 minutes

a. Measured on a 4 processor UltraSparc, at 248MHz.
b. On a hypothetical, 300MHz TriMedia 64-bit CPU, not taking into account design, fabrication, compilation.

Clearly, the 10^{15} design points of a full exhaustive search is not feasible using any approach, let alone the 10^{41} of the naive calculation. We see a factor of around 300 between machine simulation and pseudo-simulation. Both methods are cycle-accurate when it concerns pure instruction cycles.

We also see a factor of around 300 between pseudosim and machine execution. However, the time for machine execution is shown without taking the compilation trajectory into account. If we would do this, as would also be necessary for emulation on, *e.g.*, FPGA boards, then the time would in fact exceed that of pseudosim.

7. Conclusions

We have performed a full-fledged design space exploration for the 64-bit TriMedia VLIW CPU core. The outcome of design space explorations is *not* a single optimized machine, but rather:

- An integrated DSE environment in which we can investigate “what-if” questions from the CPU64 design team, with quick turn-around, based on quantitative results.
- A range of machines, each with different Area-Performance trade-offs. These serve both as candidate machines, *and* as a reference frame in which to compare the outcome of other experiments.
- Detailed knowledge from quantitative data, *e.g.*, recommendations and design rules for the placement of functional units.

Furthermore, the use of fast, pseudo-simulation techniques, combined with the described exploration approach, has brought the feasibility of design space exploration for the CPU64 within reach.

8. Acknowledgements

The authors would like to thank the many people who have contributed to making the design space exploration possible, in particular those who provided and optimized the benchmark application code, and those who provided the toolset and support libraries for the CPU64. The engineers and architects at TriMedia are acknowledged for their valuable input and inspiration.

9. References

[1] Frans Sijstermans, “Trimedia CPU64 Architecture”, *Proceedings Microprocessor Forum ‘98*, San Jose, California, Oct.12-15, 1998.

[2] J.T.J van Eijndhoven and F.W. Sijstermans, “Novel Multimedia Instruction Capabilities in VLIW Media Processors”, *Proceedings Hot Chips 10 conference*, Palo Alto (CA), 16 - 19 aug. 1998.

[3] J.T.J. van Eijndhoven et al., “TriMedia CPU64 architecture”, *these proceedings*

[4] Bart Kienhuis, *Design Space Exploration of Stream-based Dataflow Architectures*, Ph.D. thesis, Technical University of Delft, ISBN 90-5326-029-3, January 29, 1999.

[5] A.K. Riemens et al., “TriMedia CPU64 Application Domain and Benchmark suite”, *these proceedings*

[6] E.J.D. Pol, J.T.J. v. Eijndhoven, B. Aarts et al, “TriMedia CPU64 application development environment”, *these proceedings.*

[7] *Introduction to the Nelsis CAD Framework*, DIMES Design and Test Centre, Delft University of Technology, 1995.

[8] Pieter van der Wolf, *CAD Frameworks*, Kluwer Academic Publishers, ISB 0-7923-9501-8.