Transistor Count and Chip-Space Estimation of SimpleScalar-based Microprocessor Models*

Marc Steinhaus*, Reiner Kolla‡, Josep L. Larriba-Pey§
Theo Ungerer+, Mateo Valero§

*Fakultät für Informatik
Universität Karlsruhe, D-76128 Karlsruhe, Germany

‡Fakultät für Mathematik und Informatik
Universität Würzburg, D-97070 Würzburg, Germany

+Institut für Informatik
Universität Augsburg, D-86159 Augsburg, Germany

§Departament d'Arquitectura de Computadors
UPC, 08034 Barcelona, Spain

June 1, 2001

Abstract

This paper proposes a chip space and transistor count estimation tool, which receives its input from the baseline architecture and the configuration file of the microarchitecture performance simulator sim-outorder of the SimpleScalar Tool Set. The estimation tool yields a pre-silicon chip space and transistor count estimation and allows to compare different microprocessor configurations with respect to their potential chip space requirements. The estimation method, which is the basis of our tool, is validated by configuration parameters of a real processor yielding a transistor count and a chip space estimation that is very close to the real processor numbers.

1 Introduction

Evaluation of new microprocessor techniques is done by microarchitecture performance simulators based on a conceptual (register-transfer) level. Typical microarchitecture performance simulations compare the performances of hardware-enhanced models with that of a baseline model. For instance the performances of simultaneous multithreaded (SMT) processor models is compared with a single-threaded processor model applying the same configuration parameters. However, such a comparison is not usually fair, because the models often differ in their hardware requirements.

*Research partially supported by the Improving the Human Potential Programme, Access to Research Infrastructures, under contract HPRI-1999-CT-00071 "Access to CESCA and CEPBA Large Scale Facilities" established between The European Community and CESCA-CEPBA.

The resources of the baseline model should be adjusted such that the same chip space or the same transistor count is covered as in the hardware-enhanced models. Otherwise, only statements about the scaling of the regarded microarchitecture technique with respect to performance should be considered fair.

Processor performance simulators generate an instructions per cycle (IPC) count by running benchmark programs on a simulated machine. Several processor performance simulator tools are used in industry (e.g. for PowerPC [1] and for Alpha processors [17]) and research (e.g. the SimpleScalar Tool Set [2], the Karlsruhe Simultaneous Multithreaded (SMT) Multimedia Simulator KSMS [18], Tango [5], EDS [7], and the NETCARE simulators [11]). The SimpleScalar Tool Set [2] is the performance simulator for superscalar processors that is most frequently used in research currently.

Typically the simulator is based on an instruction set architecture (ISA) of a real processor architecture and a baseline microarchitecture that models a specific processor with a certain level of detail. Some parts of the microarchitecture are parametric and can be configured by a configuration file. None of the above-mentioned simulators is able to generate a chip-space or transistor count estimation of the simulated processor configurations.

This paper focuses on the implementation of a hardware complexity estimation tool based on the transistor count and chip-space requirements of the simulated microarchitecture. Our target is to allow a cost/benefit analysis in combination with the performance simulation. The tool takes information from the configuration file and makes assumptions on the baseline microarchitecture implemented in the performance simulator tool. It is implemented as a Microsoft Excel spread sheet and can be easily adapted to all kind of microarchitecture performance simulators. We choose the sim-outorder simulator of the SimpleScalar Tool Set as target of our estimation tool. Our tool is publicly available at the URL *http://www.informatik.uni-augsburg.de/lehrstuehle/info3/research/complexity/*.

There exists very scarce literature on transistor count or chip space estimation of whole processors. Most performance estimations are published without hardware complexity accounts. Some papers include transistor count estimations without explanation (e.g. [15]), some give rough transistor count estimations (e.g. [8]), and a few papers include detailed complexity estimations, in particular Palacharla, Jouppi, and Smith [13] and [14]. Burns and Gaudiot [4] perform a first step in our direction by estimating the layout area of simultaneous multithreaded (SMT) processors. They identify, which layout blocks are affected by SMT, determine the scaling of chip space requirements, and compare SMT versus single-threaded processor space requirements by scaling a R10000-based layout to 0.18 micron technology. Our own model is much more detailed. Otherwise detailed chip space or transistor count estimations are published only for single processor components, but not for full processor microarchitectures.

The next section describes the principal estimation methodology. Section 3 shortly presents the sim-outorder simulator of the SimpleScalar Tool Set and section 4 details of our specific sim-outorder directed estimation tool. Section 5 discusses the problems with the sim-outorder directed estimation and section 6 the validation of the estimation tool. The paper ends with the conclusions.

2 Estimation Methodology

When a new processor architecture is under development, its performance/area ratio must be projected to the future technology. Therefore it makes not much sense to measure area directly by mm^2. Mead and Conway [10] proposed the use of λ as the measure unit. λ is defined as half of the minimum feature size, e.g. 1 mm^2 in 0.5 micron technology equals 16 million λ^2. λ makes it easy to compare the area of designs even if they are actually fabricated in different technologies, and to project the chip area to future technologies. Therefore we will use λ^2 as area unit.

The only way to reach an accurate prediction of the chip area is a complete design in terms of λ based fabrication mask data, which is unrealistic in early stages of the development of a chip. However, the chip area can be roughly determined by assuming a transistor density *TransDens* and by estimating the number of transistors, i.e. an expected transistor count *TransCount*. The chip area is then estimated by

$$LayoutArea = TransCount/TransDens.$$

The transistor density reflects the routing overhead and differs from component to component of a microarchitecture. In general the density is higher for very regular structured full custom cells like memories, and is lower for more unstructured units like controllers or functional units.

When developing a new processor, the only area/transistor count information we have is the scanty experience from existing proprietary processors. In the best case, a processor is publicly specified by its minimum feature size, its overall die area, overall transistor count, and microarchitecture features like the number, size and properties of on chip memories, buffers, queues and registers, number and type of functional units etc. Among this bunch of components that account for the whole chip area, we identify three subsets:

1. Components which are very sensitive to architectural decisions.

   Typical examples are all SRAM-based memory components, like data and instruction caches, register sets, renaming tables, branch prediction buffers, etc. The size and the number of ports of such memory components are critical for many architectural decisions, however the implementation of these components is well known. *Analytical* formulas can be developed that estimate the area or transistor count from size, word width and port parameters.

2. Components whose complexity is invariant under most architectural decisions.

   Such components are the functional units. Their internal design is so complex and there are so many degrees of freedom to implement them that it is difficult to develop an analytical model to predict area or transistor count. However, from the architectural point of view, their chip space and transistor requirements are essentially determined by the word width and it is mainly the number and types of functional units that varies from architecture to architecture. These components are candidates for an *empirical estimation* method.

3. All other components.

   The rest of components have many possible implementations but are also quite sensitive to architectural decisions. Examples of these are control logic, arbiters, bus systems, clock generation etc. For some of these components the architectural parameters have an influence on their area, e.g. the instruction issue unit strongly

depends on the issue width of the processor. In these cases, the transistor count and density of these units can be determined empirically and be related to the architectural parameter.

All other components, which cannot be realistically estimated neither empirically nor analytically, are only a small portion of the whole processor. They can be simply modeled as a certain percentage for the area overhead with respect to the area, estimated from the assessable part of the processor.

Our estimation tool is able to derive a transistor count and chip area estimation with significant accuracy [22], using inputs from the SimpleScalar Tool Set. To estimate the chip space and the amount of transistors we use an analytical method for memory-based structures like register files or internal queues and an empirical method for logic blocks like control logic and functional units.

Analytical model

Multiport SRAMs occur within many components of processors, like register files, rename tables, branch prediction buffers etc., so that an analytical model for multiported SRAMs is a versatile part within the development of analytical models for many components of a microarchitecture. The transistor count model of a memory-based component $TransMemComp$ is composed by the SRAM cell for a single bit $TransSRAMBit$ times the number of bits $Size$.

$$TransMemComp = TransSRAMBit \cdot Size$$

Based on a typical schematic for SRAM cells (see [22] for full detailed account), we derive a formula for the transistor count assuming four transistors per bit cell and additionally one transistor per read and two transistors per write port of the bit cell. Let RP be the number of read only ports and WP be the number of read/write ports, then

$$TransSRAMBit(RP, WP) = RP + 2 \cdot WP + 4$$

To calculate the chip-space of a memory based structure we estimate the basic area of a bit cell that is increased in height and width by the number of ports. We derive an analytical model of the SRAM core from the layout sketch of an SRAM bit cell considering the following parameters that account for the area size of the SRAM bit cell:

- the width $WtDataWidth$ contributed by one data line,
- the height $WtAddressWidth$ contributed by one address line,
- the basic width of the SRAM bit cell $SRCBasicW$, and
- the basic height of the SRAM bit cell $SRCBasicH$.

The formulas for area calculation assume that the address wires are connected to the smaller side $SRCBasicH$ of the memory cell and the data wires to the broader side $SRCBasicW$. A read port needs a data wire and an address wire, whereas a write port needs one address and two data wires. The width of the small side of a SRAM bit cell cell $SRCBasicCH$ is computed as

$$SRCBasicCH = SRCBasicH + (WP + RP) \cdot WtAddressWidth$$

and for the broader side $SRCBasicCW$ as

$$SRCBasicCW = SRCBasicW + (2 \cdot WP + RP) \cdot WtDataWidth$$

The area of the SRAM bit cell $AreaSRAMBit$ is then given by

$$RegSize = SRCBasicCW \cdot SRCBasicCH$$

and the area of the SRAM memory component $areaSRAMComp$ by

$$AreaSRAMComp = AreaSRAMBit \cdot Size$$

where $Size$ is characterized by the number of rows r times the number of columns c of SRAM bit cells of the whole SRAM. We use this analytical area model for multi-ported SRAMs and assume in coherence with [9] a value of 8λ for $WtDataWidth$ and $WtAddressWidth$, 32λ for $SRCBasicW$ and 24λ for $SRCBasicH$.

Empirical model

Our empirical approach for the non memory-based components of the processor model, i.e. the functional units and logic blocks, works as follows. We measure the component areas in floor-plans of existing processors and estimate the areas of the components in λ^2. To estimate the transistor count of such processor components with our tool, we calculate the average transistor density of non memory-based components of real processors again by measuring floor-plans and by additional information about the transistor count of the measured components.
Transistor densities should be empirically determined for all non memory-based components. However, this is difficult, because of the scarce information in literature. Therefore we apply in our current estimation tool a single transistor density $LogTransDens$ for all non memory-based components.
We measured transistor densities of the non memory-based area of HP-PA 8000 [12] and Fujitsu SPARC64 [6] yielding transistor densities of $td_{HP-PA8000} = 703.3$ transistors per 1 million λ^2 and $td_{SPARC64} = 520.9$ transistors per 1 million λ^2. The arithmetical mean of both, $td_P = 612.1$ transistors per 1 million λ^2, is currently applied as our estimated transistor density $LogTransDens$ for non memory-based components. The model should be refined, if better basic transistor densities were found.

3 The SimpleScalar Tool Set

The SimpleScalar Tool Set [2], developed by Todd Austin at the University of Wisconsin at Madison, is a collection of tools for the simulation of superscalar processors at an abstract level. We choose the sim-outorder, the most detailed of the provided simulators, as the basis of our estimator. The base ISA version of sim-outorder is based on the MIPS IV [16], but it is modified depending on the specific research requirements yielding many different versions of sim-outorder. Out-of-order execution is controlled by the Register Update Unit (RUU) ([3], [20]). Real processors that implement the MIPS ISA, e.g. the MIPS R10000 [23] and R12000, are considerably different in their microarchitecture.
Fig. 1 shows the pipeline of sim-outorder. Instructions are fetched from the I-cache in the *Fetch* stage. The configured branch prediction method influences the quality of the fetched instructions. The *Dispatch* stage, decodes and dispatches the instructions to the register update unit (RUU). The RUU consists of a pool of reservation stations, a register rename unit and a reorder buffer. There is a separate load/store queue (LSQ) for the load and store instructions. The *Scheduler*, respectively the *MemoryScheduler*, issues instructions to the execution units. Load and store instructions are separated into two sub-instruction queues each, one for address computation, the other one for

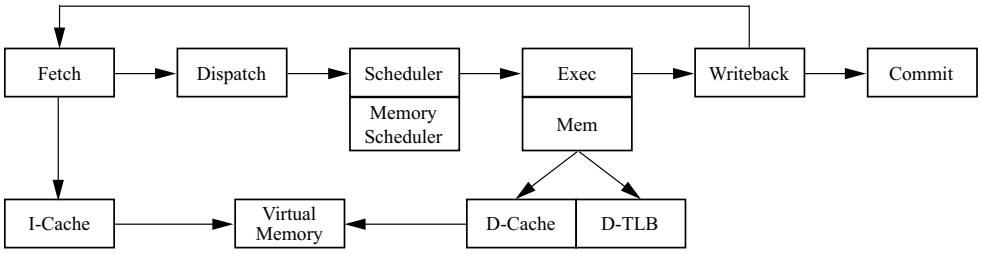

Figure 1: Pipeline of the sim-outorder simulator of the SimpleScalar Tool Set [2]

the memory access. The entries in the LSQ contain a reference to the respective entry in the RUU. The RUU and the LSQ are organized as ring buffers. In the *Writeback* stage the results are written by the result bus of the execution unit to the RUU. The *Commit* stage updates the registers and memory.
The processor components of sim-outorder can be configured by many parameters. The parameters are read from a configuration file at program start.
A set of those parameters will be used by the estimator that we present here. Those are listed below with their respective functions and their names in our estimation tool

- $-fetch : ifqsize < size >$ fetch bandwidth $FWidth$
- $-decode : width < insts >$ decode bandwidth $DecWidth$
- $-issue : width < insts >$ issue bandwidth $IssWidth$
- $-ruu : size < insts >$ size of RUU buffer $RuuSize$
- $-lsq : size < insts >$ size of LSQ buffer $LsqSize$
- $-res : ialu < num >$ number of integer units $NuIntAlu$
- $-res : imul < num >$ number of complex integer units $NuIntMul$
- $-res : fpalu < num >$ number of floating-point units $NuFpAlu$
- $-res : fpmul < num >$ number of complex floating-point units $NuFpMul$
- $-res : memports < num >$ number of ports (read and write ports not distinguished) to access the first level cache $NuMemPort$
- $cache :< name >< config >$ specifies first and secondary D- and I-caches by parameter $< name >$ set to $dl1$, $dl2$, $il1$, and $il2$. $< config >$ is specified as $< nsets >:< bsize >:< assoc >:< repl >$ meaning:
  - $< nsets >$ number of cache sets $CacheSet$
  - $< bsize >$ cache block size in bytes $CacheBlckSize$
  - $< assoc >$ associativity $CacheAss$
  - $< repl >$ replacement strategy; we only define the LRU strategy $CacheLru$

- *tlb* :< *name* >< *config* > specifies D- and I-TLBs by the parameter < *name* > set to *dtlb* or *itlb*. < *config* > is specified as < *nsets* >:< *bsize* >:< *assoc* >:< *repl* > meaning:
  - < *nsets* > number of TLB sets $TlbSet$
  - < *assoc* > associativity $TlbAss$
  - < *repl* > TLB replacement strategy; we only define the LRU strategy $TlbLru$
  - < *bsize* > page size in bytes $TlbPageSize$
- *bpred* < *type* > branch prediction strategy; we only support the bimodal *bimod* strategy with < *size* > number of 2-bit entries in the branch history table BHT. The BHT is combined with the BTAC and its size is set by the parameter $BtbBtacSize$ in our estimator.

4 Complexity Estimation for the SimpleScalar Tool Set

In this section we show in detail, how our tool estimates the transistor counts and chip areas of the memory-based components, i.e. registers, caches, TLBs, the register update unit RUU and the load/store queue LSQ as well as the non memory-based components of sim-outorder.

4.1 Registers

The baseline processor of sim-outorder contains two register sets: an integer register set and a floating-point register set. Register width is 32 bits each (also for floats!). The number of architectural registers in the register sets is defined as 32 in sim-outorder. However, the number of physical registers is left open, i.e. potentially unlimited. Here, we allow to define the total number of general-purpose registers by $NuReg$ and the total number of floating-point registers by $NuFpReg$.
The number of bits per register set is computed from the number of registers $NuReg$ respectively $NuFpReg$ multiplied by 32 (bits per register). If other than 32 bit registers could be specified, this can easily be changed in the estimator. For integer and for floating-point registers the following formulas are applied:

$$RegBit = 32 \cdot NuReg$$
$$RegFpBit = 32 \cdot NuFpReg$$

The number of read ports differs from the number of write ports. The number of write ports is determined in sim-outorder by the commit stage of the pipeline [3], which can write back as many results to the architectural registers as the instruction fetch stage can fetch, i.e.

$$RegWP = Fwidth \text{ and } FpRegWP = Fwidth$$

The execution units read directly from the registers. Therefore the number of read ports depends on the number of execution units that are able to access the specific register set. Integer registers store integer values and load and store addresses. Each integer unit may read two operands simultaneously from the registers.
We additionally assume for the estimator that the load/store units may read as many values from the registers as memory ports are available. Stores read also the store

addresses besides the values from the registers, we therefore assume that each memory port of sim-outorder needs two read ports for the integer register set.
Concerning the floating-point register set, floating-point stores read only values from the floating-point register set, because the addresses are stored in integer registers. We therefore assume only one read port per memory port for the floating-point registers. Read port calculation is performed by the following formulas:

$$RegRP = (NuIntMul + NuIntAlu + NuMemPort) \cdot 2$$

and

$$FpRegRP = (NuFpMul + NuFpAlu) \cdot 2 + NuMemPort$$

The transistor count calculation is done by (in section 2 we set: 4 transistors per register bit cell, $TransWP = 2$, and $TransRP = 1$):

$$TransReg = RegBit \cdot (RegWP \cdot TransWP + RegRP \cdot TransRP + 4)$$

and

$$TransFpReg = FpRegBit \cdot (FpRegWP \cdot TransWP + FpRegRP \cdot TransRP + 4)$$

The area of a register set (integer or floating-point) is computed in coherence with the formulas in section 2 as

$$\begin{aligned} & RegSize(RegBit, RegRP, RegWP) \\ = \; & ((RegRP + 2 \cdot RegWP) \cdot WtDataWidth + SRCBasicW) \\ & \cdot((RegRP + RegWP) \cdot WtAddressWidth + SRCBasicH) \cdot RegBit \end{aligned}$$

4.2 Caches

The size and number of transistors for all four types of caches in sim-outorder are calculated in the same way. The cache bits are calculated from the parameters for number of sets, associativity, and cache line size. The line size is specified in bytes, therefore the formula must be multiplied by eight. The data bits in a cache are calculated by:

$$CacheBit = CacheSet \cdot CacheBlckSize \cdot CacheAss \cdot 8$$

To compute the number of tag bits $CacheTagBit$ the address width of the cache line address is necessary. First, the index bits $IdxBit$ are computed:

$$IdxBit = \lceil Log_2(CacheSet) \rceil$$

Second, the offset bits $OffBit$ that identify the byte in a cache line are computed:

$$OffBit = \lceil Log_2(CacheBlckSize) \rceil$$

Additional bits may be necessary for the replacement strategy. We only support the Least Recently Used (LRU) strategy. The bits for the LRU strategy $LruBit$ are determined by

$$LruBit = Log_2(CacheAss)$$

All tag bits are computed by the formula ($AddWidth$ determines the address width):

$$CacheTagBit = (AddWidth + LruBit - IdxBit - OffBit) \cdot CacheSet \cdot CacheAss$$

Simplescalar does not distinguish between read and write ports in the D-cache. Only the number of memory ports can be specified and this is modeled by the parameter $NuMemPort$. The sim-outorder allows all memory ports to access the D-cache simultaneously. The data cache is not banked, therefore the number of D-cache ports depends on the number of memory ports (to load/store units or memory), potentially resulting in a large growth of the data cache area.

The number of ports of the I-cache should be at least as high as the instruction fetch bandwidth. The sim-outorder assumes a 4 byte instruction width. Thus the number of I-cache ports are computed by:

$$CachePort = \lceil (Fwidth \cdot 4)/CacheBlckSize \rceil$$

For transistor count estimation of caches we start from the assumptions about the transistor count of each SRAM cell, and each read and each write port. Because the number of read ports equals the number of write ports for all caches of sim-outorder, we count the number of transistors per cache bit as follows (note, in section 2 we set: $TransSRBit = 4$, $TransWP = 2$, and $TransRP = 1$):

$$TransCacheBit = TransSRBit + (TransRP + TransWP) \cdot CachePort$$

and the overall transistor count of caches:

$$TransCache = (CacheBit + CacheTagBit) \cdot TransCacheBit$$

The cache area is computed from the areas for the single SRAM bit cells (see sections 2 and 4.1). First the height and width of each cache cell are computed as:

$$CacheCH = SRCBasicH + (CacheRP + CacheWP) \cdot WtAddressWidth$$
$$CacheCW = SRCBasicW + (CacheRP + 2 \cdot CacheWP) \cdot WtDataWidth$$

The full cache area is estimated as:

$$CacheSize = (CacheBit + CacheTag) \cdot CacheCH \cdot CacheCW$$

4.3 Translation Look-aside Buffers TLBs

Translation look-aside buffers (TLBs) are internally similar to caches. Therefore transistor count and chip space is estimated with the same methods used for caches. We again assume that the number of read ports is the same as the number of write ports. In the following we only demonstrate the bit calculation, the number of ports is calculated corresponding to the port numbers in caches.

TLBs are required for fast transformation of virtual to physical addresses. The TLB stores the necessary parts of the virtual address $TlbTag$ and the parts of the physical address $TlbAdd$, which are necessary to identify the memory page. The address bits used to address a byte within the page are identical for the physical and virtual addressing. This page offset $PageOff$ is computed as

$$PageOff = Log_2(TlbPageSize)$$

To identify a set within the TLB, part of the virtual address is used. This part must not be stored in the TLB and is computed as:

$$TlbIdx = Log_2(TlbSet)$$

Assuming a LRU replacement strategy we need

$$TlbLru = Log_2(TlbAss)$$

number of bits for its implementation.

Thus the transistor count of a TLB is the sum of the following formulas:

$$TlbTag = (AddWid + TlbLru - PageOff - TlbIdx) \cdot TlbAss \cdot TlbSet$$

and

$$TlbAdd = (AddWid - PageOff) \cdot TlbAss \cdot TlbSet$$

4.4 Register Update Unit RUU

The RUU is the central component that controls out-of-order execution of instructions in the SimpleScalar toolset. The RUU contains the pool of reservation stations, the rename register set, and the reorder buffer. An entry in the RUU contains both source

operands including register tags and other tag bits, the result with register tag and ready bit, the program counter, functional unit identification number, a dispatched bit and an executed bit. Altogether, an RUU entry consists of the following fields (see [3]):

- Source1: Register tag, Ready bit, Content
- Source2: Register tag, Ready bit, Content
- Result: Register tag, Ready bit, Content
- Program counter PC, Functional unit ID
- Dispatched bit, Executed bit

The number of transistors and the area of the memory-based part of the RUU is calculated analytically analogous with the register set calculation. We describe the bit and port calculation only.
The number of bits stored in the RUU is calculated as product of the number of entries and the sum of bits of each entry. Source operand and result parts are of the same size. The *Content* contains an operand of 32 bits and the *Ready bit* is one bit only. Thus, an (source or result) register part is:

$$RegPart = RegTag + 1 + 32$$

The size of a register tag depends on the number of registers and is computed as follows:

$$RegTag = \lceil Log_2(NuReg + NuFpReg) \rceil$$

The size of the program counter is not defined in sim-outorder, but can be set in the estimator by the parameter $PCBit$. The number of bits for the Functional unit ID $FuID$ is calculated as:

$$FuID = \lceil Log_2(NuIntAlu + NuIntMul + NuFpAlu + NuFpMul + NuMemPort) \rceil$$

We assume a load and store unit for each memory port.
For the tags *Dispatched* and *Executed* we need one bit each. Altogether an RUU entry needs the following number of bits:

$$RuuBit = 3 \cdot RegPart + PCBit + FuID + 2$$

It is essential for the port calculation that the RUU as central memory structure for execution control is accessed by many processor components simultaneously. The issue unit, called *Scheduler* in sim-outorder, accesses the RUU for issuing of instructions to the execution units. This unit needs as many read ports as the issue bandwidth $IssWidth$ of the processor. The bandwidth of the $Writeback$ stage (buffering results in the RUU) cannot be configured. Because all execution units potentially yield results simultaneously, we assume as many result buses as execution units and therefore we need as many write ports to the RUU as the number of execution units. The execution units also access the RUU, because the RUU stores operands as well as results. Because both operands of an instruction are stored in parallel in the RUU, only a single additional read port per execution unit is necessary.
The write-back stage writes results from the execution units to the RUU, therefore the RUU needs as many write ports. The Commit unit, which writes back the results in the registers, also has to access the RUU in reading fashion. The Commit unit can commit as many instructions as the fetch unit can fetch, therefore we assume $Fwidth$ as bandwidth of the Commit unit resulting in the same amount of read ports to the RUU. The decode unit, called Dispatch stage in SimpleScalar, needs as many write ports as

instructions can be dispatched to the RUU each cycle ($DecWidth$). This results in the following formulas:

$$RuuRP = IssWidth + NuFU + Fwidth$$
$$RuuWP = NuFU + DecWidth \text{ with}$$
$$NuFU = NuIntAlu + NuIntMul + NuFpAlu + NuFpMul$$

Such an implementation of the RUU results in a number of problems that are discussed in section 5.

4.5 Load/Store Queue LSQ

The Load/Store Queue LSQ is also implemented as ring buffer. Load and Store instructions are separated in two parts. The first part is the address stored in the RUU, the second part is the loaded data or data to be stored to the memory. The second part of a load or store instruction is included in the LSQ. The transistor count and area calculations are analogous to the register calculations of the RUU estimations. Therefore, we only describe the bit and port calculations.
The number of LSQ bits is determined by the number of LSQ entries and the number of bits per entry. A LSQ entry consists of the memory address $AddWidth$, the 32 bit of data, the one bit ready tag, which shows when the instruction is ready for issue, a reference to the corresponding RUU entry $RuuIdBit$, a one bit tag if the instruction is issued, and another bit tag if the instruction is done. Altogether an entry contains the following fields:

- Load/store address, Data, Ready
- RUU entry ID
- Dispatched bit, executed bit

The size of the reference to the RUU is calculated as

$$RuuIdBit = \lceil Log_2(RuuSize) \rceil$$

and the number of LSQ bits as

$$LsqBit = LsqSize \cdot (32 + 3 + AddWidth + RuuIdBit)$$

For the port calculation of the LSQ we assume a read and a write port for each memory port $NuMemPort$, because the load/store units access the LSQ to read the memory address ($Load/storeaddress$) and read or write the data.
The results of the address calculations are written to the LSQ by the Writeback stage. Therefore the number of write ports of the LSQ is increased by the write-back bandwidth, which is assumed to be the same as the number of execution units $NuFU$.
The instruction decode unit stores the initial entries in the LSQ. Therefore the number of write ports is increased by the decode bandwidth $DecWidth$.
The commit stage writes the values from the LSQ to the architectural registers, which increases the number of LSQ read ports by the commit bandwidth, which is equivalent to the fetch bandwidth $Fwidth$ [3].
The instruction issue unit for store instructions, in SimpleScalar called $Scheduler$, issues the instruction in the LSQ to the load/store units. It can be assumed that the number of load and store instructions that can be issued is equivalent to the ports available. Because the memory scheduler must examine how many instructions are there with ready bit set, we assume for each memory port $NuMemPort$ one more read port. Altogether the number of read ports of the LSQ is

$$LsqRP = NuMemPort \cdot 2 + Fwidth$$

and the number of write ports

$$LsqWP = NuMemPort + NuFU + DecWidth$$

As for the RUU, we did not examine optimized and more realistic implementations.

4.6 Logic-based Processor Components

In this section we give an account of the different components that are based on logic circuits rather than memory structures:

The complexity of the instruction fetch unit depends on the fetch bandwidth. We assume a linear growth [1] of the transistor count and area requirement with the fetch bandwidth. Therefore, the formula we yield for area estimation is:

$$FetchUSize = FUBasicArea \cdot FWidth$$

and for the transistor count is:

$$FetchUTrans = FUBasicTrans \cdot FWidth$$

The Dispatch unit decodes the instructions and dispatches the instructions to the RUU, respectively the LSQ and the RUU. Its size depends on the number of decoded and dispatched instructions per cycle. The estimation of the area that we assume is

$$DecUSize = DecBasicArea \cdot DecWidth$$

and the transistor count that we assume is

$$DecUTrans = DecBasicTrans \cdot DecWidth$$

The *Scheduler* unit issues instructions of the RUU and the LSQ to the execution units respectively load/store units. We assume that the complexity depends on the issue bandwidth. The area formula that we assume is

$$IssUSize = IssBasicArea \cdot IssWidth$$

and the transistor count that we assume is

$$IssUTrans = IssBasicTrans \cdot IssWidth$$

The area and transistor count of the functional units depends on the basic areas and the number of each type only. For each functional unit type $FUType$ we assume

$$FuTypeSize = NuFuType \cdot FuTypeBasicSize$$

for area and

$$FuTypeTrans = NuFuType \cdot FuTypeBasicTrans$$

for transistor count.

The $Writeback$ stage writes results of execution units to the RUU. The bandwidth cannot be specified in SimpleScalar. Therefore we assume it to be equivalent to the number of execution units $NuFU$. Its area is computed as

$$WbSize = NuFU \cdot WbBasicArea$$

and its transistor count is computed as

$$WbTrans = NuFU \cdot WbBasicTrans$$

The *Commit* stage writes results from the RUU to the registers. Its bandwidth cannot be configured and corresponds to the instruction fetch bandwidth $FWidth$. The area is calculated as

$$ComSize = ComBasicArea \cdot FWidth$$

and transistor count is calculated as

$$ComTrans = ComBasicTrans \cdot FWidth$$

[1] This is a rough estimate, but better formulas can easily be build into updated versions of the estimation tool.

5 Difficulties with the Complexity Model for SimpleScalar

The microarchitecture of the baseline SimpleScalar processor differs strongly from real existing processor microarchitectures. In particular the RUU for out-of-order execution is not implemented anywhere.
As has been elaborated in section 4.4, the RUU, as central memory structure for execution control, is accessed by many processor components simultaneously. This causes in particular problems in terms of the port definition. The issue unit, called *Scheduler* in SimpleScalar, accesses the RUU to issue instructions to the execution units. This unit needs as many read ports as the issue bandwidth $IssWidth$ of the processor. The bandwidth of the *Writeback* stage (buffering results in the RUU) cannot be configured. Because all the execution units potentially yield results simultaneously, we assume as many result buses as execution units and therefore we need as many write ports to the RUU as number of execution units. The execution units also access the RUU because the RUU stores operands as well as results. Because both operands of an instruction are stored in parallel in the RUU, only a single additional read port per execution unit is necessary.
The write-back stage writes results from the execution units to the RUU, therefore the RUU needs as many write ports. The Commit unit, which writes back the results in the registers, also has to access the RUU in reading fashion. The Commit unit can commit as many instructions as the fetch unit can fetch, therefore we assume $Fwidth$ as the bandwidth of the Commit unit resulting in the same amount of read ports to the RUU. The decode unit, called Dispatch stage in SimpleScalar, needs as many write ports as instruction can be dispatched to the RUU every cycle ($DecWidth$).
Implementing the RUU this way yields a large increase of the number of ports and therefore a large increase in hardware complexity, in particular for the area. A real implementation of the RUU is not known to us. However, the complexity evaluation of realistic implementation variants of the RUU would result in restrictions not taken into account in a corresponding performance simulator. We choose to provide the base version of sim-outorder in the basic version of the estimation tool (see URL *http://www.informatik.uni-augsburg.de/lehrstuehle/info3/research/complexity/*) and instruct all the users on how to adapt this part to their specific sim-outorder version.
Another problem with sim-outorder based estimation concerns its unclear register specification assuming an unlimited number of physical registers. Here, we allow to define the total number of general-purpose registers and of floating-point registers separately, and initialize this to 32 each in the basic version of the estimation tool. However, we assume that a limited number of physical registers may yield other performance results when using sim-outorder with benchmark programs.

6 Validation of the Complexity Model

A validation of the full baseline processor of sim-outorder is not possible (see last section). However, the estimation models are similar to the Karlsruhe SMT Multimedia Simulator KSMS ([18], [19]) which can be validated, because it closely models a real processor, the PowerPC 604 [21]. This gives also some plausibility to the SimpleScalar models. Partial models of processor components of the SimpleScalar directed estimator

model can be evaluated and are included in the validation of the KSMS.
To validate the estimation tool, we estimate a processor similar to a PowerPC by configuring the KSMS simulator-directed version of the estimation tool such that the configuration is similar to the PowerPC 604. We analytically calculate transistor count and chip area of the memory-based components and we measured the PowerPC 604 floor plan (see [21]) for chip space estimation of non memory-based units. The transistor density from HP-PA 8000 [12] and Fujitsu SPARC64 [6] is used to generate a transistor count equivalent for the non memory-based components. The validation is particularly interesting for the transistor count of non memory-based components estimated by the transistor density and for the calculated transistor counts and chip space values for the memory-based components. We do not know transistor counts for the different components of PowerPC 604. We only know the total number of transistors, 3.6 million, and the total chip space 196 mm^2 in 0.5 micron technology of 1994 [21].
Our estimated total transistor count of 3.58 million transistors differs only slightly from the 3.6 million transistors stated for the PowerPC 604. The estimated total area of PowerPC 184,8 mm^2 is also close to the real area, 196 mm^2.

7 Conclusions

This paper proposes a chip space and transistor count estimation tool, which receives its input from the baseline architecture and the configuration file of a microarchitecture performance simulator. We choose the sim-outorder of the SimpleScalar Tool Set, because this is most widely used. Our estimation tool gives a pre-silicon complexity estimation and allows to compare different microprocessor configurations with respect to their hardware complexity. To estimate the chip space and the amount of transistors, we use an analytical method for memory-based structures like register files or internal queues, and an empirical method for logic blocks like control logic and functional units. While the analytical method works with sufficient accuracy, the empirical estimation of logic blocks suffers from insufficient data about chip areas and transistor densities of control logic blocks and functional units from real processors. Here the estimation tool should be provided by better data which can easily introduced by changing single parameters in the tool (see the read-me section of *http://www.informatik.uni-augsburg.de/lehrstuehle/info3/research/complexity/*).
The tool takes information from the configuration file and makes assumptions on the baseline microarchitecture of sim-outorder. Unfortunately the microarchitecture of sim-outorder does not resemble any real processor and contains several unrealistic assumptions, in particular too many read/write ports in the RUU and unlimited number of physical registers of the general-purpose and the floating-point register set. Therefore the estimation method cannot be validated for the sim-outorder estimation tool. However, its Karlsruhe SMT Multimedia Simulator KSMS directed version is validated configuring the estimator with the parameters of the PowerPC 604 processor, which yielded a transistor count and a chip space estimation that is very close to the real processor numbers of the PowerPC 604.

8 References

[1] Anderson William *An Overview of Motorola's PowerPC Simulator Family.* Communications of the ACM, Vol. 37, No. 6, June 1994, 64-69.

[2] Burger Doug, Austin Todd M. *The SimpleScalar Tool Set, Version 2.0.* http://www.cs.wisc.edu/ mscalar/simplescalar.html

[3] Burger Doug *Hardware Techniques to Improve the Performance of the Processor Memory Interface.* PhD Dissertation, University of Wisconsin-Madison, 1998.

[4] Burns James, Gaudiot Jean-Luc *Quantifying the SMT Layout Overhead - Does SMT Pull Its Weight?* The Sixth International Symposium on High-Performance Computer Architecture (HPCA-6), Toulouse, France, Jan. 8-12, 2000, pp. 109-120.

[5] Davis Helen, Goldschmidt Stephen R., and Hennessy John *Multiprocessor Simulation and Tracing Using Tango* 1991 International Conference on Parallel Processing, II-99-107.

[6] Fujutsu *The SPARC64 Processor.* HAL Computer Systems, Inc.

[7] Krishnan Venkata, Torrellas Josep *A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors.* PACT, Paris, France, Oct. 1998, pp. 286-293.

[8] Lipasti Mikko H., Shen John P. *Superspeculative Microarchitecture for Beyond AD 2000.* IEEE Computer, September 1997, pp. 59-66.

[9] Lopez David, Llosa Josep, Valero Mateo, and Ayguade Eduard *Widening Resources: A Cost-effective Technique for Aggressive ILP Architectures.* The 31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998, pp. 237-246.

[10] Mead C., Conway L. *Introduction to VLSI Systems.* Series in Computer Science. Addison-Wesley, Reading, MA, 1980.

[11] NETCARE: NETwork-computer for Computer Architecture Research and Education. http://www.ece.purdue.edu/netcare

[12] Kumar Ashok *The HP-PA8000 RISC CPU: A High Performance Out-of-Order Processor.* Hot Chips VIII, August 1996, pp. 9–20.

[13] Palacharla Subbaro, Jouppi Norman P., and Smith James E. *Complexity-Effective Superscalar Processors.* 28th Annual International Symposium on Computer Architecture, 1997, pp. 206-218.

[14] Palacharla Subbarao, Jouppi Norman P., and Smith James E. *Quantifying the Complexity of Superscalar Processors.* Technical Report CS-TR-96-1328 http://www.cd.wisc.edu/tsr.html

[15] Patt Yale N., Patel Sanjay J., Evers Marius, Friendly Daniel H., and Stark Jared *One Billion Transistors, One Uniprocessor, One Chip.* IEEE Computer, September 1997, pp. 51-57.

[16] Price Charles *MIPS IV Instruction Set, revision 3.1.* MIPS Technologies, Inc., Mountain View, CA, January 1995.

[17] Reilly Matt, Edmondson John *Performance Simulation of an Alpha Microprocessor.* IEEE Computer, May 1998, 50-58.

[18] Sigmund Ulrich *Entwurf und Evaluierung mehrfädig superskalarer Prozessortechniken im Hinblick auf Multimedia.* PhD Thesis, University of Karlsruhe, May 2000 (in German).

[19] Sigmund Ulrich, Marc Steinhaus, and Theo Ungerer *On Performance, Transistor Count and Chip Space Assessment of Multimedia-enhanced Simultaneous Multithreaded Processors.* Fourth Workshop on Multithreaded Execution, Achitecture and Compilation (MTEAC-4) in association with 33rd International Symposium on Microarchitecture (MICRO-33), December 10, 2000.

[20] Sohi, Gurindar S., Vayapeyam, Sriram *Instruction Issue Logic for High-performance, Interruptable Pipelined Processors.* 25 Years of the International Symposia on Computer Architecture (Selected Papers), 1998, pp. 329–336.

[21] Song Peter S., Denman Marvin, and Chang Joe *The PowerPC 604 RISC Microprocessor.* IEEE Micro, Oktober 1994.

[22] Steinhaus Marc, Kolla Reiner, Larriba-Pey Josep L., Ungerer Theo, Valero Mateo *Transistor Count and Chip-Space Estimation of Simulated Microprocessors.* Research report UPC-DAC-2001-16, UPC, Barcelona, Spain.

[23] Yeager Kenneth C. *The Mips R10000 Superscalar Microprocessor.* IEEE Micro, April 1996, pp. 28–40.