Process Simulation in the Browser: Porting ViennaTS using WebAssembly

Xaver Klemenschits, Paul Manstetten, Lado Filipovic, and Siegfried Selberherr
Institute for Microelectronics, TU Wien, Gußhausstraße 27-29/E360, 1040 Wien, Austria
Email: klemenschits@iue.tuwien.ac.at

***Abstract*—We introduce a client-side browser application for high performance process simulation. The codebase is taken from ViennaTS, an open-source C++ process simulation engine, and completed with JavaScript based components, such as an editor to configure the simulation parameters and a result viewer. The C++ codebase is ported to the browser by compiling to a portable standard for an abstract instruction set: WebAssembly. We demonstrate the capabilities and performance of the application by performing several configurable simulations, including the emulation of the fabrication process of a stacked nanosheet field effect transistor. The simulations conducted in the browser application are only slower by a factor of 1.6 to 3.6 compared to native, single-thread simulations. Therefore, WebAssembly presents a promising format for portable and widely accessible high performance process simulations.**

I. Introduction

The introduction of three-dimensional metal oxide semiconductor field effect transistors (MOSFETs), such as FinFETs and nanowire transistors, has created the need for intricate fabrication techniques incorporating new materials and processes. As the fabrication of devices for advanced technology nodes becomes increasingly complex and costly, technology computer aided design (TCAD) is more and more important to investigate possible new fabrication techniques. Especially software development has benefited strongly from open source initiatives, creating productive global platforms for innovation. Although open-source tools for process simulation exist, their wide and active adoption and thus the creation of creative development platforms is often limited by a narrow application focus pursued by a small user group. As large numbers of users are beneficial to open source projects, deployment, accessibility, and visibility are highly important for a wider acceptance of such tools.

With the advent of WebAssembly (WA) [1], an open standard describing a binary format expressing assembly-like instructions for a virtual processor, a framework for portable high performance code was created. The WA design rationale is to establish a portable executable format which can be compiled quickly for any target system, in a fast single-pass fashion. The WA framework therefore allows client-side browser applications to be built using existing C/C++ codebases, enabling portable high performance applications to be deployed and accessed at any scale without requiring backend investments.

ViennaTS [2], an open-source, high performance process simulator written in C++, was ported to the browser, since it offers a wide array of simple as well as highly complex process models describing modern semiconductor fabrication steps. Here, we present the results of compiling ViennaTS using WA combined with a user interface to edit parameters and view simulation results in a unified application [3], which is easily deployable at scale.

II. Process Simulation using ViennaTS

ViennaTS [2] is an open-source, C++ based process simulator which supports two- and three-dimensional simulations of semiconductor fabrication processes. The materials and their interfaces are represented with a sparse-field level-set framework discretised on a Cartesian grid [4]. Large numbers of materials can therefore be represented accurately with minimal memory requirements. The level-set is stored in a hierarchical run length encoded (HRLE) data structure, optimised for level-set operations, such as re-distancing, velocity extension, and advection on a sparse data set. A set of advanced physical etching and deposition models is available, which include particle transport at feature-scale and intricate chemical models for the description of surface reactions. Particle transport at feature-scale is modelled using a Monte Carlo ray tracing approach via an explicit representation of the surface with partially overlapping disks [5]. Accelerated geometric models to emulate various process steps are also available, which advance the material interfaces solely by geometric considerations, offering efficient, albeit less accurate, process descriptions. Therefore, many different processing techniques with varying accuracies can be combined and tuned to match a wide field of applications.

III. WebAssembly

WebAssembly is an open standard for a *virtual instruction set architecture (ISA)*[1]. Basic components of an ISA are supported data types, an instruction set (for control flow, arithmetic/logic, and memory operations) and its encoding. The advantages of the *virtual* ISA of WA over a *native* ISA of a specific processor is that it has a portable hardware-, platform-, and language-independent design, which aims to execute in a sandboxed memory-environment with near-native speed. The interfacing/interoperation with the environment provided by the execution platform has a simple universal design and is not limited to web-platforms (i.e., the JavaScript engine of a browser). Further design goals are the ability to efficiently decode, validate, and compile the instructions of a WA program on the execution platform.

Since 2017 WA support has been widely adopted across most major browsers and JavaScript engines, and about 85% of currently installed browsers support it [6]. When compared to JavaScript, the execution performance and performance predictability of WA profits from the fact that all types are

978-1-7281-0940-4/19/$31.00

statically declared and programs can be compiled ahead-of-time and the execution is not interrupted by a garbage collector.

Additionally to the integration of WA support in all major browsers and JavaScript engines, there is also active development of WA-runtimes for desktop applications and efforts are ongoing to standardise the communication between WA and it's embedding platform with a WebAssembly System Interface (WASI)[7]. Due to the aforementioned universal interfacing of WA with its environment WASI, each platform implements its own interfacing strategy. The WASI aims to allow for a modular specification to provide a portable modular WA-interfacing.

The motivation for WA is due in part to the success of its predecessors *NaCl*[8] and especially *asm.js*[9]. The JavaScript library *asm.js* can even be seen as a blueprint for the WA standard, since it allowed native C/C++ to JavaScript compilation, which resulted in performant execution due to the use of a limited subset of JavaScript language features. Only the features which can be accessed by the ahead-of-time optimisation of all common JavaScript engines are part of this subset. Realising the potential of this approach, browser vendors implemented optimisations specifically for *asm.js* in their JavaScript engines, increasing the performance further.

The infrastructure for compiling C/C++ for a web-browser is currently actively developed in *emscripten* [10] and *LLVM*[11]. This framework aims to keep the necessary changes to the C/C++ codebase at a minimum by providing all implementations of commonly used interfaces to the environment, e.g., by providing a virtual file system and tailored standard libraries to map system calls to common browser engine instructions.

Currently, the main limitations, when porting an existing C/C++ codebase to WA, include a maximum memory of 4GB (WA has a 32bit address space), no wide support for C++ threads/OpenMP, and limited support for dynamic linking.

In the following, we describe the portable application built around our ViennaTS WA module. Porting the C++ codebase to be compiled to WA with *emscripten* required 570 lines to be added or modified, from a total of 28,986 lines of the entire project. Therefore, the change required to port this existing high performance C++ codebase to WA only corresponds to about 2% of the total code base.

IV. In-Browser Application

Fig. 1 provides an overview of the system components typically involved on a desktop computer when using ViennaTS to simulate a process step. After preparation of the process parameters in an *Editor*, *ViennaTS* (`vts.exe`) is started by providing the parameter file (`par.txt`) which references the initial geometries (`geo.vtk`). The results (`res.vtk`) can then be visualised by loading them into a *Result Viewer*.

Analogously to Fig. 1, Fig. 2 provides an overview of the components necessary for the In-Browser Application: The user interface is executed in the *UI Thread* of the browser and contains an *Editor* and a *Result Viewer*, which are both pure JavaScript applications. The *Web Worker*, executed in a different thread than the user interface, loads *ViennaTS* (`vts.wasm`) and compiles it for the current host system in a single pass. When the simulation is started, the parameter

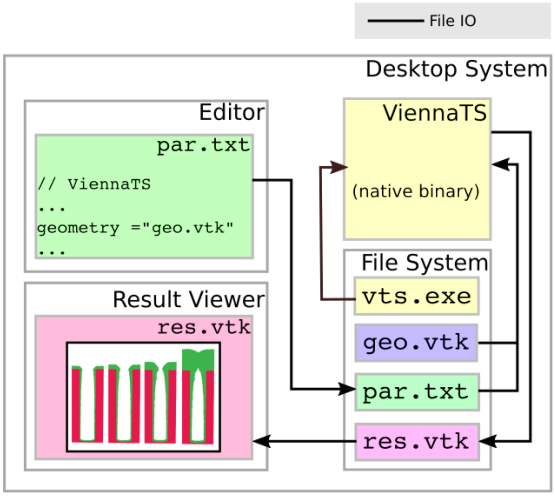


Fig. 1: Components typically involved in the preparation, execution and inspection of simulations conducted with the native desktop version of ViennaTS [2].

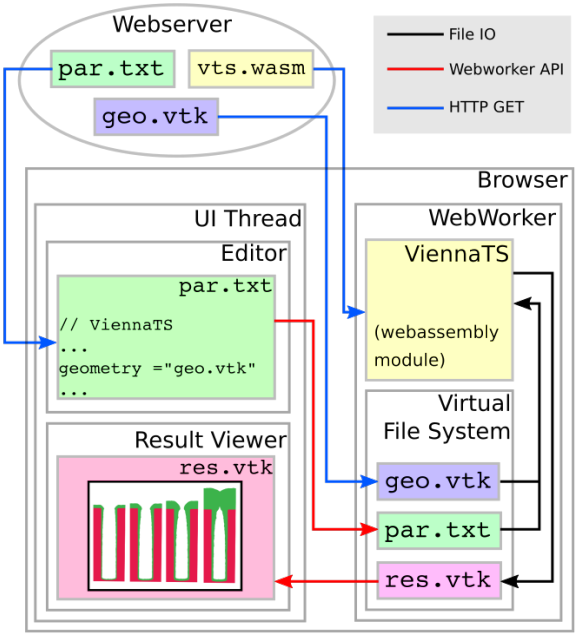


Fig. 2: Components involved in conducting a simulation with the ViennaTS in-browser application available at hpcwasm.github.io/viennats.

file (`par.txt`) is transferred to the *Web Worker* and stored in its *Virtual File System*. The initial geometries (`geo.vtk`), referenced in `par.txt`, are then fetched from the *Webserver* by the *Web Worker*. Once all required files are available, the *Web Worker* executes ViennaTS which writes the simulation results (`res.vtk`) to the *Virtual File System*. As soon as the results have been written, they are automatically transferred to the *UI Thread* and visualised in the *Result Viewer*.

The full user-interface can be seen in Fig. 3 which shows the most important parts needed to start a simulation. The "Settings" section allows the user to load and manipulate existing parameter files using an *Editor*. Once the user has populated the parameter file, the simulation can be started

using the controls in the ”Simulation” section, which also shows text-based simulator output and simulation progress. In the ”Results” section, the generated output meshes are shown in the *Result Viewer*, providing basic mesh inspection and manipulation functionality as well as the ability to download the results.

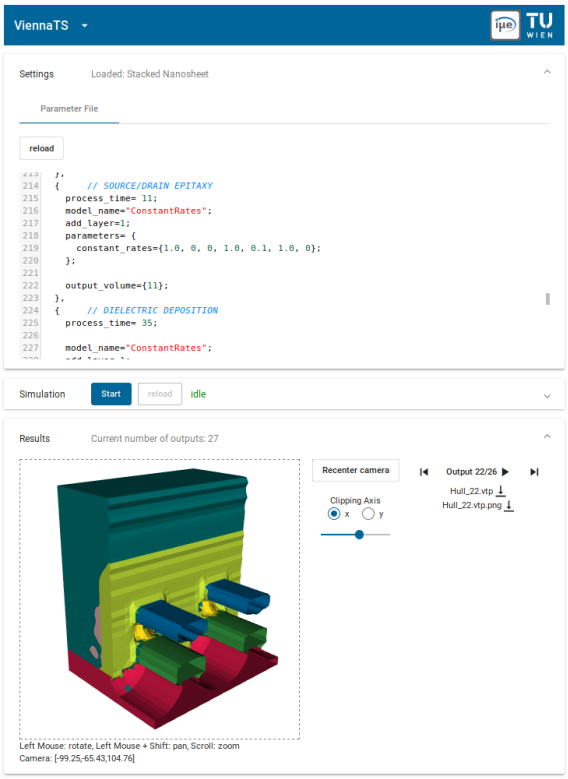


Fig. 3: Graphical User Interface of the application. The *Editor* in the ”Settings” section is used to create a parameter file used as input for ViennaTS. The ”Simulation” section provides additional console output when expanded and the ”Results” section shows the output geometries as they become available in the *Result Viewer* and provides mesh manipulation tools.

V. Benchmarks & Performance Evaluation

In order to evaluate the performance differences between the WA and native versions of ViennaTS, numerous benchmark simulations were conducted. Some of these apply only emulations, advancing material interfaces purely by geometric parameters, while others use Monte Carlo (MC) ray tracing methods to model the particle transport inside a reactor at the feature-scale and provide chemical models to describe surface reactions. All benchmark simulations are available and can be executed online at [3].

Table I shows the different execution times obtained on a desktop computer and a smartphone for several single-threaded simulations. On the desktop, the runtime performance gap between native and browser execution varies between a factor of 1.6 to 3.6, depending on the simulation. The runtime performance gap between native execution and mobile browsers is even larger. The mobile version is slower by a factor of 10.0 to 18.6. The performance gap is caused firstly by optimisation differences, since two separate compilation stages are necessary for WA, and secondly by the more complex memory management of a browser compared to an operating system. The latter results in a larger runtime gap for simulations which are more memory intensive. On the other hand, Monte Carlo enabled simulations (labelled MC in Table I), which are rather computationally than memory intensive, show smaller runtime gaps.

Simulation	Native	Firefox	Chrome	Mobile
Geom. Hole Etch	6.97	16.97 (2.4)	20.46 (2.9)	128.62 (18.5)
Phys. Hole Etch(MC)	11.95	30.46 (2.5)	42.88 (3.6)	188.11 (15.7)
Geom. Air Gap Depo	12.27	30.10 (2.5)	36.08 (2.9)	228.56 (18.6)
Phys. Air Gap Depo (MC)	12.99	21.23 (1.6)	22.58 (1.7)	130.01 (10.0)
Geom. HAR Etch	5.26	12.77 (2.4)	15.90 (3.0)	92.88 (17.7)
Phys. HAR Etch (MC)	12.00	31.21 (2.6)	39.91 (3.3)	189.64 (15.8)
Geom. Deposition	6.54	15.57 (2.4)	19.84 (3.0)	121.51 (18.6)
Phys. Deposition(MC)	34.65	52.34 (1.5)	58.61 (1.7)	–
Stacked Nanosheet	9.61	25.19 (2.6)	27.79 (2.9)	–
Avg. Perf. Gap	–	2.3	2.8	16.4

TABLE I: Runtime (without output) comparison for the showcased examples of process emulations (Geom.) and physical simulations (Phys.) on a desktop computer (Linux; Intel i7-3770 processor) natively (*Native*) and in the browser (*Chrome 73.0*, *Firefox 66.0*). Additionally, *Mobile* benchmarks (*Firefox Mobile 66.0*) were conducted on a smartphone (Android; Kirin 650 processor). The relative runtime gaps to the native execution are shown in brackets. Models labelled (MC) use physical models with Monte Carlo ray tracing methods.

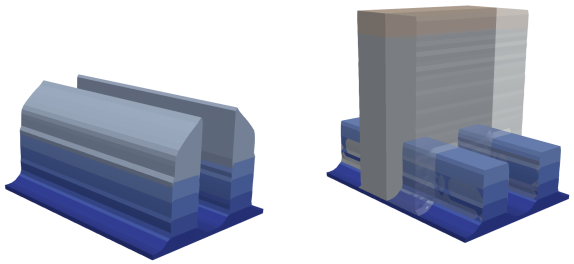
VI. Stacked Nanosheet Fabrication Process

In order to highlight the full capabilities and performance of the ported simulator, the fabrication process of an advanced-node stacked nanosheet FET was emulated starting from a blank SOI wafer. Since this geometry is comparably large, the high memory requirements result in a relatively large runtime gap, even exceeding the capabilities of the tested mobile browser. Therefore, all models used to create the final structure were purely geometric and MC models were not applied. These models and their input parameters are shown in Table II.

Experimental data from [12] and [13] were used to calibrate the process steps, resulting in a geometry which matches a typical structure in use today. Fig. 4 shows the resulting geometry at the most important process steps: Starting from a blank SOI wafer, Si, SiGe, and Si layers are deposited epitaxially. These layers are then patterned to fins, using self aligned double patterning, creating the structure shown in Fig. 4a. After removing the mask, a dummy gate is created by depositing Poly-Si isotropically over the fins. As can be seen in Fig. 4b, the resulting Poly-Si is then patterned to the required gate dimensions [14]. Hf is deposited around the gate to form insulating spacers between the gate and source/drain regions. Directional etching is then applied to generate the required vertical spacers, as shown in Fig. 4c. The fins extending out from under the gate and spacer are removed to expose the bottom Si layer. This Si layer is subsequently used as a seed material for the epitaxial growth of Si, forming the source and drain by connecting the two silicon channels of each fin, which results in the structure shown in Fig. 4d. Next, an interlayer dielectric is deposited and the surface is levelled using chemical mechanical planarisation (CMP). After the removal of the dummy gate, the SiGe layer separating the two

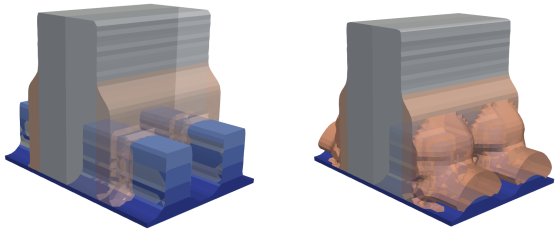
Physical Process	ViennaTS Model	Geometric Parameters
Si Epitaxy	ConstantRates	Si=+7nm
SiGe Epitaxy	ConstantRates	SiGe=+8nm
Si Epitaxy	ConstantRates	Si=+7nm
Fin Mask	Mask	Add predefined mask
SADP Mask Growth	ConstantRates	SADPMask=+15nm
Pattern SADP Mask	DirectionalRates	SADPMask=(0,0,-20nm)
Fin Mask Removal	BooleanOp	Remove Fin Mask
Fin Patterning [a]	DirectionalRates	Si/SiGe=-30nm, Spacer=-15nm
SADP Mask Removal	BooleanOp	Remove SADPMask Material
Dummy Gate Deposition	ConstantRates	PolySi=+55nm
Dummy Gate CMP	Planarization	PolySi
Gate Mask	Mask	Add predefined mask
Dummy Gate Patterning [b]	DirectionalRates	PolySi=-90nm
Gate Mask Removal	BooleanOp	Remove Gate Mask
Gate Spacer Deposition	ConstantRates	Hf=+12nm
Gate Spacer Patterning [c]	DirectionalRates	Hf=(0,0,-35nm)
Remove Fins at S/D	DirectionalRates	Si/SiGe=-20nm
S/D Epitaxy [d]	ConstantRates	Si=+11nm
ILD Deposition	ConstantRates	ILD=+35nm
ILD CMP	Planarization	ILD
Dummy Gate Removal	ConstantRates	PolySi=-80nm
NW Release [e]	ConstantRates	SiGe=-10nm
Gate Dielectric Deposition	ConstantRates	Hf=+2
Gate Metal Deposition	ConstantRates	TiN=+4nm
Gate Electrode Deposition	ConstantRates	W=+20nm
Final CMP [f]	Planarization	All materials

TABLE II: Summary of the geometric models used to create the structure, as shown in Fig. 4. The model "ConstantRates" isotropically deposits(+) or etches(-) the respective material. "DirectionalRates" deposits/etches the material only in a specified direction. "Mask" adds a predefined mask geometry, "Planarization" flattens the topology at a certain height, and "BooleanOps" is used here to completely remove materials. Superscript letters refer to the subfigures in Fig. 4. The exact parameters used in the simulation can be found at hpcwasm.github.io/viennats/#/simulation/stackednanosheet.



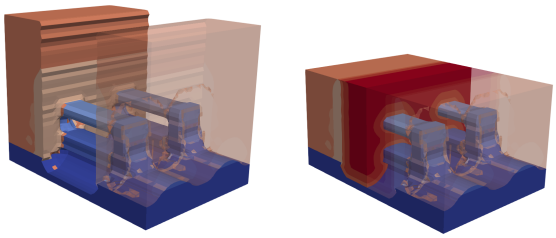
(a) Epitaxial growth and double patterning.

(b) Dummy gate patterning.



(c) Spacer formation and patterning.

(d) Source/drain epitaxy.



(e) Channel release.

(f) Final geometry after gate material deposition.

Fig. 4: Full stacked nanosheet process emulated in the browser using ViennaTS for WebAssembly. The figures show the geometry after the process steps described in each caption.

Si nanowires is etched away leaving the nanowires suspended in the air, as shown in Fig. 4e. Finally, the gate dielectric (HfO_2), the gate metal (TiN), and the gate contact material (W) are deposited to create the final structure shown in Fig. 4f.

VII. Conclusion

The C++ codebase of ViennaTS was compiled to WebAssembly and executed in the browser. By doing so, it was shown that porting complex C++ codebases to a performant portable format can be achieved with few changes to the original code. WebAssembly therefore presents a promising platform for the deployment of scientific software.

Native performance cannot be matched due to the technical limitations of portable code execution. However, we could show that the average performance gap for the process simulator ViennaTS is only a factor of 2.3 for Firefox and 2.8 for Chrome compared to native desktop execution. Since all computations are carried out on the client-side in the browser, this enables portable high performance process simulations, accurately describing complex state-of-the-art fabrication techniques commonly applied in the semiconductor industry.

References

[1] WebAssemblyCommunityGroup(W3C). Webassembly Specification. [Online]. Available: https://webassembly.github.io/spec/

[2] O. Ertl *et al.* ViennaTS - The Vienna Topography Simulator. [Online]. Available: https://github.com/viennats/viennats-dev

[3] P. Manstetten and X. Klemenschits. ViennaTS - Webassembly Port. [Online]. Available: https://hpcwasm.github.io/viennats

[4] R. T. Whitaker, "A Level-Set Approach to 3D Reconstruction from Range Data," *International Journal of Computer Vision*, vol. 29, no. 3, pp. 203–231, 1998.

[5] O. Ertl and S. Selberherr, "Three-Dimensional Topography Simulation using Advanced Level Set and Ray Tracing Methods," in *2008 International Conference on Simulation of Semiconductor Processes and Devices*. IEEE, 2008, pp. 325–328.

[6] A. Deveria. caniuse.com. [Online]. Available: https://caniuse.com/#search=webassembly

[7] WebAssemblyCommunityGroup(W3C). Webassembly System Interface. [Online]. Available: https://github.com/WebAssembly/WASI

[8] Google. Google Native Client. [Online]. Available: https://chromium.googlesource.com/native_client/src/native_client.git

[9] A. Zakai, Mozilla. asm.js. [Online]. Available: http://asmjs.org

[10] A. Zakai, "Emscripten: an LLVM-to-JavaScript Compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. ACM, 2011, pp. 301–312.

[11] LLVM Developer Group. LLVM. [Online]. Available: http://llvm.org

[12] H. Mertens *et al.*, "Gate-All-Around MOSFETs based on Vertically Stacked Horizontal Si Nanowires in a Replacement Metal Gate Process on Bulk Si Substrates," in *2016 Symposium on VLSI Technology*. IEEE, 2016, pp. 1–2.

[13] N. Loubet *et al.*, "Stacked Nanosheet Gate-All-Around Transistor to Enable Scaling beyond FinFET," in *2017 Symposium on VLSI Technology*. IEEE, 2017, pp. T230–T231.

[14] S. Barraud *et al.*, "Tunability of Parasitic Channel in Gate-All-Around Stacked Nanosheets," in *2018 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2018, pp. 21.3.1–21.3.4.