Automatically Exploring Computer System Design Spaces

Bobby R. Bruce
bbruce@ucdavis.edu
University of California, Davis
Davis, California, USA

ABSTRACT

While much research has focused on using search to optimize software, it is possible to use these same approaches to optimize other parts of the computer system. With decreased costs in silicon customization, and the return of centralized systems carrying out specialized tasks, the time is right to begin thinking about tools to optimize systems for the needs of specific software targets. In the approach outlined, the standard Genetic Improvement process is flipped with source code considered static and the remainder of the computer system altered to the needs of the software. The project proposed is preliminary research into incorporating grammar-based GP with an advanced computer architecture simulator to automatically design computer systems. I argue this approach has the potential to significantly improve the design of computer systems while reducing manual effort.

CCS CONCEPTS

• **Software and its engineering** → *Search-based software engineering*; • **Computer systems organization** → *Architectures.*

KEYWORDS

search, genetic improvement, computer architecture

ACM Reference Format:
Bobby R. Bruce. 2022. Automatically Exploring Computer System Design Spaces. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3520304.3534021

1 THE VISION

Source code is a human-readable language for describing the desired functionality of a computer system. Compilers, the operating system with its kernel and drivers, the ISA, the chips, the microarchitecture, the logic gates, and everything else, all exist to serve the needs of what is defined in code. While software engineers have to occasionally tweak hardware parameters or interface with low-level with components, the trajectory of software development has been one of increasing abstraction from the remainder of the computer system. Software engineers thereby have the privilege to choose the best computer for the job. This activity creates an implicit, invisible coordination between those who produce applications, and those who develop the remainder of the computer system. The software engineers produce required applications while the ecosystem of hardware designers, OS developers, etc. respond by ensuring the software runs as well as is possible at a tolerable price-point. In a sense, there is a human-driven evolution of computer systems with software developers providing the necessary evolutionary pressure.

Unfortunately this process is an expensive, largely human-driven affair with engineers responsible for carrying out this iterative evolutionary process. In order to improve this I ask the following question: *With just source code and inputs representative of its usage, can a optimal computer system be automatically designed?*

I believe there are signs that such a goal may be achievable. Search-based techniques, such as Genetic Improvement [4] (GI), have shown that software can be optimized for hardware targets. What I propose is the flipping of this process. Instead of seeing a host computer system as the fixed entity with software as a malleable component, why not view software as the definition of a system's behavior and modify everything else accordingly?

There are some clear economic incentives to carry out such research. As an example, a SaaS service provider may run many instances of the same software for years. It would therefore be in their interest to ensure the system this software runs is optimally designed; the highest performance for the lowest cost. They could optimize at many different levels of the computer system. At the level of silicon they could look into customization options. While historically bespoke silicon solutions were costly, the costs of design-to-tapeout has decreased substantially. SiFive go as far as offering customization of their RISV-V cores via a web-interface [1]. From here they could look into the optimal size and layout of caches and memory, and the possible inclusion of hardware accelerators. Moving into the realms of software they can tweak kernel and wider operating system parameters so the target software performs as best it can on their bespoke hardware. The compilation of the input software may also be tweaked to produce machine-level instructions optimal for the hardware. Compilation flags and the order of compiler optimizations can have considerable impact on a given workload's performance in a particular environment. Such design decisions can be made now (and in some cases are) but it is prohibitively expensive for most use-cases. This leaves many to pick general purpose solutions and accept poorer performance. I would hope, with the research outlined, we can reduce the costs involved in computer system optimization thus making it a more common endeavor for system designers.

The ultimate end-goal of what I am proposing is a framework in which software may be written independently of all other considerations with an automated process left to generate the optimal computer system for it to run. This grand goal is distant and ambitious, but we have the tools now to make headway with each step on this journey fruitful for computer architects and others who design computer systems.

2 FIRST STEPS

What do the grass roots of this end-goal look like? Thus far I have deliberately used the term "computer system" when discussing optimization as it is broad and all-encompassing, but we must start research with something more specific. Computer systems can be understood from high level interactions between nodes in a network, all the way down to nano-scale interactions on a wafer. Surveying the literature we find that work has already been carried out on the automated generation of electronic circuits [5]. While this work is worthy of note, working at this scale produces too large a search space to automatically create functional, modern computer systems in a reasonable timescale. I thereby propose work start at the level of computer architecture where we deal with pre-built, somewhat inter-changeable, components. This approach allows for large degrees of configurability while utilizing off-the-shelf components and standardized interfaces.

To optimize these components a search-based approach will be used. As using search will involve evaluating a vast number of computer systems to find a suitable solution, we must rely on simulation tools. Fortunately simulators exist at the computer architecture scale; gem5 [2] being the most notable. Therefore, in the research outlined, a search-based tool will be built atop such a simulator. The search algorithm will be responsible for producing a design, and the simulator responsible for producing the fitness of the design. As a simple first step, this tool will configure broad properties such as what processor to use, the size and nature of the cache hierarchy, and the size of the memory system. These properties will then be optimized with respect to a particular budget. A multi-objective algorithm that attempts to provide a Pareto optimal set is also possible.

Architecture simulators take configurations then run a simulation to produce statistics. These configurations are, in a sense, a language to describe a particular system. I therefore propose using Grammar-based GP [3] to build these configurations. The Grammar-based GP algorithm will construct the configuration (a description of the system in the simulator's "language") with the simulator providing the fitness. Using a grammar-based GP will allow extension or reduction of the grammar based on what aspects of the design the experimenter wishes to investigate. Unlike other approaches, a grammar-based approach allows for the introduction of sensible constraints to reduce the search space down to solutions that are desirable.

The extendable nature of this framework will allow for the incorporation of more exotic designs and configurations. Moving along from the basic first-step, it will be possible to incorporate more complex decisions such as which ISA extensions to include and what competing micro-architectural designs to use. We can also work further up the layers to toggling kernel parameters, the operating system, or compiler optimizations. It would be interesting to see how optimizations at this level could produce synergistic improvements with changes at lower levels. As many of these design considerations interact, we should expect a non-linear search space.

The engineering of this system will, in itself, involve considerable engineering effort, but there are also tooling and infrastructure problems that should be solved first. Modern simulators are frequently criticized for not producing results that reflect the systems they simulate. As the proposed framework would function by searching through a set of simulated components, we can target engineering effort to ensure these components simulate systems within an acceptable margin of error. This is a non-trivial task and will involve carefully creating simulated components with real-world equivalents on-hand to ensure they are functioning as intended. The grammar defining how components may be incorporated can also be engineered to limit combinations of components to those known to produce reasonable results. In addition to accurate results from components, we also need sensible modeling of their cost. This is necessary to optimize systems within a certain budget or to produce Pareto optimal sets of solutions. A naive first step would be to optimize while simply minimizing the number of components and, for some components, their complexity, but we will need to to move onto a more realistic model eventually.

Even if we assume accurate simulations with good cost modelling, these simulations take time to execute. For example, in the gem5 simulator a full boot of the Ubuntu operating system can take several hours. As search involves evaluating many solutions, this is problematic. We can, however, take steps to alleviate this. First, simulations within a single generation can be run independently meaning we can utilize parallelization. The second step is to incorporate simulation tricks. We can, for example, only simulate particular parts of an input program's execution and extrapolate the findings. Finally, we can develop a hierarchy of workloads in which small tests are evaluated first and may be used to filter obviously bad designs early. In essence, we can front-load evaluation with "sanity checks" that incur a low evaluation cost but allow a "fail fast" strategy for bad solutions.

3 CLOSING REMARKS

Much optimization research has been carried out regarding the optimization of software. In this paper I have outlined an approach that sees the entirety of a computer system as something that is configurable. Using the latest from the field of search-based optimization, I believe we can start building tools to design systems with respect to a target workload. With the advent of open-source ISAs such as RISC-V, the continued improvements in software portability, and customization available all the way down to the level of silicon, I believe the time is right to start considering research and development of such tools.

REFERENCES

[1] [n.d.]. SiFive Core Designer. https://www.sifive.com/core-designer. Accessed: 2022-04-05.

[2] Jason Lowe-Power et al. 2020. The gem5 Simulator: Version 20.0+. arXiv:arXiv:2007.03152

[3] Robert I McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'neill. 2010. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines* 11, 3 (2010), 365–396. https://doi.org/10.1007/s10710-010-9109-y

[4] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. 2017. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2017), 415–432. https://doi.org/10.1109/TEVC.2017.2693219

[5] Adrian Thompson. 2012. *Hardware Evolution: Automatic design of electronic circuits in reconfigurable hardware by artificial evolution.* Springer Science & Business Media.