ArchExplorer for Automatic Design Space Exploration

Growing architectural complexity and stringent time-to-market constraints suggest the need to move architecture design beyond parametric exploration to structural exploration. ArchExplorer is a Web-based permanent and open design-space exploration framework that lets researchers compare their designs against others. The authors demonstrate their approach by exploring the design space of an on-chip memory subsystem and a multicore processor.

Veerle Desmet
Ghent University

Sylvain Girbal
Thales Research and Technologies

Alex Ramirez
Augusto Vega
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

Olivier Temam
Inria Saclay

In designing a processor architecture, architects usually rely on a trial-and-error process in which intuition and experience drive the creation and selection of appropriate designs. However, as architecture complexity and the number of possible architecture options increases, experience and intuition might not be the best drivers for architecture design decisions. The current trend toward multicores will likely only exacerbate this problem. The cache mechanisms comparison by Gracia-Perez et al. illustrates this concern by suggesting that the progress of research might not always be regular over time, in large part because the current methodology does not emphasize the comparison of research results.[1]

This article presumes an architecture complexity tipping point at which human insight would be more productive if systematically combined with architecture design-space exploration (see the "Related Work in Design-space Exploration" for other research in this area). The Gracia-Perez et al. study highlighted the risks of insufficient comparison and reproducibility, but did not propose a proper framework for systematic design-space exploration beyond modular simulation for easier reuse. Although modular simulation, as proposed in SystemC (http://www.systemc.org), Liberty,[2] MicroLib,[1] United Simulation Environment (Unisim),[3] and Asim,[4] facilitates reuse and comparison, it assumes that all or many researchers will adopt the same simulation platform, which is realistic only within a controlled environment, such as has occurred for Asim at Intel.

ArchExplorer focuses on the practical challenges that prevent a researcher from performing a broad exploration and fair comparison of architecture ideas, especially the time and complexity involved in reimplementing other researchers' works.[5] ArchExplorer, developed as part of the European SARC integrated project, is a framework for an open and permanent exploration of the architecture design space. This solution takes the form of a website rather than a traditional simulation environment. Instead of asking a researcher to find, download, and run the simulators of competing mechanisms, ArchExplorer provides a

0272-1732/10/$26.00 © 2010 IEEE Published by the IEEE Computer Society

Related Work in Design-space Exploration

Several research works attempt to generalize design-space exploration and provide frameworks for architecture exploration, although most do not bring design-space exploration beyond parameter exploration.

The Magellan framework for multicore exploration embeds power/area measurement and statistical exploration techniques, and exposes a large range of multicore parameters.[1] ReSP enables the exploration of architectures composed of transaction-level SystemC components, as well as hardware/software trade-offs.[2]

Palermo focuses on the exploration of embedded systems, particularly heuristics, to converge rapidly to Pareto-optimal configurations (performance, power, delay).[3] Similarly, Pimentel proposes the Sesame framework for design-space exploration in the context of systems on chip.[4] This framework uses multiple abstraction levels to speed up the exploration.

Emer et al. present a notable exception to parametric-only design-space exploration.[5] They create efficient branch predictors by decomposing branch prediction algorithms into elementary primitives and then composing them, creating new branch predictors.

For both architectural and compiler exploration, many recent works, such as Ipek et al.[6] and Lee et al.,[7] discuss building statistical models using machine-learning techniques, showing that it is possible to converge rapidly to good solutions in a huge design space.

References

1. S. Kang and R. Kumar, ''Magellan: A Search and Machine Learning-Based Framework for Fast Multicore Design Space Exploration and Optimization,'' *Proc. Design, Automation, and Test in Europe* (DATE 08), ACM Press, 2008, pp. 1432-1437.
2. G. Beltrame, L. Fossati, and D. Sciuto, ''Resp: A Nonintrusive Transaction-Level Reflective MPSOC Simulation Platform for Design Space Exploration,'' *IEEE Trans. CAD of Integrated Circuits and Systems,* vol. 28, no. 12, 2009, pp. 1857-1869.
3. G. Palermo, C. Silvano, and V. Zaccaria, ''Multi-Objective Design Space Exploration of Embedded Systems,'' *J. Embedded Computing,* vol. 1, no. 3, 2005, pp. 305-316.
4. A.D. Pimentel, C. Erbas, and S. Polstra, ''A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels,'' *IEEE Trans. Computers,* vol. 55, no. 2, Feb. 2006, pp. 99-112.
5. J.S. Emer and N.C. Gloy, ''A Language for Describing Predictors and Its Application to Automatic Synthesis,'' *Proc. 24th Ann. Int'l Symp. Computer Architecture,* ACM Press, 1997, pp. 304-314.
6. E. Ipek et al., ''Efficiently Exploring Architectural Design Spaces via Predictive Modeling,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, 2006, pp. 195-206.
7. B.C. Lee and D.M. Brooks, ''Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, 2006, pp. 185-194.

remote environment in which researchers can upload their own simulators and compare them against previously uploaded mechanisms or compose a new mechanism. The continuous exploration both enables the exploration of a huge design space over a long period and progressively builds over time a large database of results that will further speed any future comparison.

After the researcher uploads a mechanism, the entire exploration process is automatic—from plugging the mechanism into an architecture target, to retuning the compiler for that target, statistically exploring the design space, and publicly disseminating the exploration results.

A framework for open and continuous exploration

Figure 1 shows the overall methodology. In short, a researcher adapts a custom simulator to make it compatible with the ArchExplorer environment and uploads the simulator together with a specification of valid and potentially interesting parameter ranges. The mechanism is added immediately to the continuously explored design space, and ArchExplorer statistically selects and explores the architecture design points. For each architecture design point, the Web-based infrastructure automatically retunes the compiler for a meaningful comparison with other architecture design points and recompiles the benchmarks accordingly. After ArchExplorer simulates the set of benchmarks for the design point, it accumulates performance results in the database and automatically updates and disseminates the ranking of the mechanisms on the website.

Automatically composing architectures

ArchExplorer does not require that researchers use a given common simulation platform; that approach would be unrealistic.

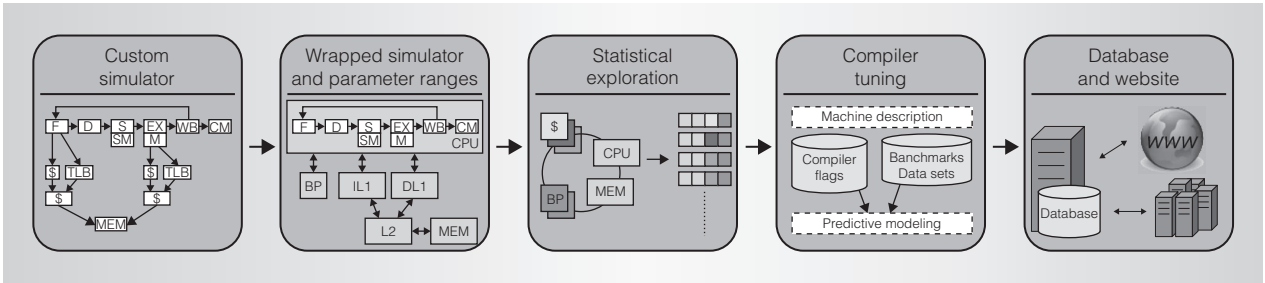

Figure 1. Overview of the ArchExplorer system. A researcher adapts a custom simulator and uploads this simulator with a specification of parameter ranges. The rest of the process—exploring the design space, including compiler tuning, and disseminating results on the website—is fully automated.

In many cases, researchers have invested significant effort in building or familiarizing themselves with a simulation platform, so it is unlikely that they would want to discard that effort, even for the sake of comparison. However, a simulator part cannot be integrated into another simulation environment without modification. The task often appears more daunting than it is in practice. In addition, modular simulation environments such as Asim, Liberty, and SystemC, or modular simulators such as M5,[6] SimFlex,[7] and Gems,[8] are increasing users' sensitivity to the productivity benefits of modularity, which makes simulators more easily compatible with the ArchExplorer approach.

Architecture communications interfaces. Both architecture parts—the uploaded hardware block and the server-side architecture (for example, an uploaded data cache and a server-side processor)—must be compatible at the hardware level. For that purpose, both parts should agree on a set of input and output control and data signals. This set of signals forms a communication interface—the *architecture communications interface* (ACI), analogous to software-level application programming interfaces (APIs). Data caches require two such ACIs: a processor/cache ACI and a cache/interconnect (bus, network on chip) ACI. Because both interfaces correspond to a processor/memory interface, only a single interface is required. Table 1 shows the interface's specifications.

Potentially, any data cache mechanism that implements this ACI can be plugged automatically into a processor that supports it.

Table 1. Processor/memory interface.

Interface fields	Details
`Address`	Bidirectional, 32 bits
	Memory request address
`Data`	Bidirectional, path width
	Data for read and write requests
`Size`	Bidirectional, $\log_2$(max(*#bytes*)) bits
	Request size in bytes
`Command`	Processor → cache, 3 bits
Processor/L1	Request type (`read, write, evict, prefetch`)
L1/L2	Request type (`read, write, evict, prefetch, readx, flush`)
Processor/memory	Request type (`read, write, evict, prefetch, readx, flush`)
`Cacheable`	Processor → cache, 1 bit
	Whether or not the requested address is cacheable

As a result, all such compatible data cache mechanisms can be automatically explored, assuming that software-level compatibility issues, discussed later, have been resolved. For example, using the processor/memory interface, arbitrarily deep cache hierarchies can be composed with interface-abiding caches, as Figure 2a shows.

ACIs raise two main questions:

- Must a new ACI be designed for each new hardware block variation?
- Which hardware blocks, typically studied in architecture research, are eligible to an ACI definition?

For most data cache mechanisms, the innovations proposed are internal to the block. These innovations have little or no impact on the interface with the processor

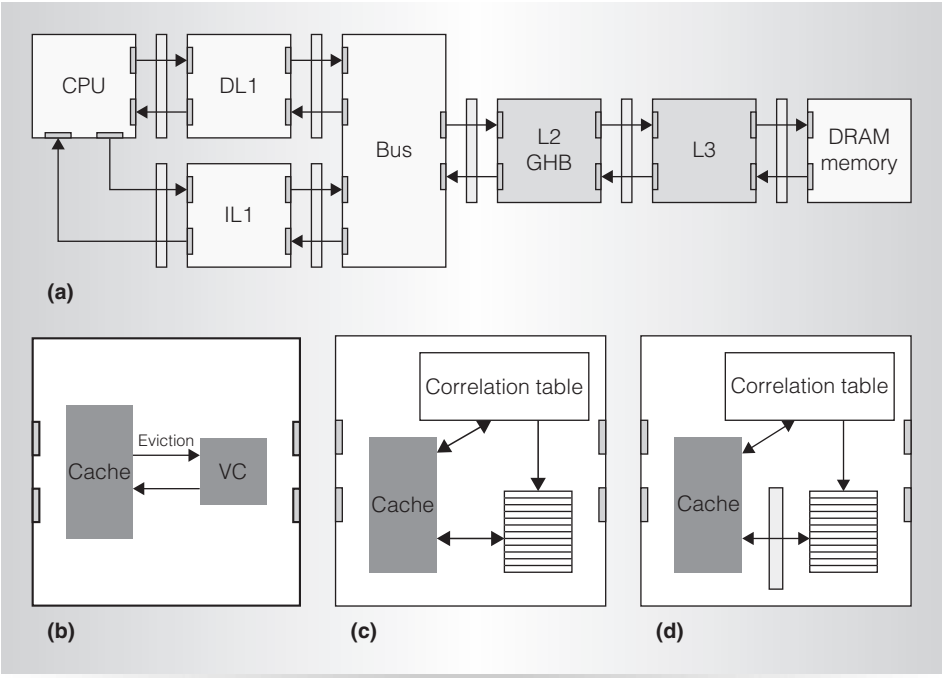

Figure 2. Modular simulation environment of ArchExplorer. Composing architectures (a) and fine-grained compositions: victim cache (VC) (b), time keeping (TK) (c), and prefetch buffer application communications interface (ACI) (d). (GHB: global history buffer)

and with the memory in many, though not all, cases. Of the nine data cache mechanisms listed in Table 2, all the mechanisms but one, dead-block correlating prefetcher (DBCP),[9] are compatible with the interface in Table 1, which corresponds to the standard data cache interface. DBCP builds a history trace of load/store requests extracted from the commit stage of an out-of-order processor. The corresponding signals/wires are not part of the ACI derived from the standard data cache. Because DBCP cannot be plugged as is, the ACI must be extended to accommodate this mechanism. However, a large subset of the data cache mechanisms can share the same ACI with no special effort. When a mechanism is not compatible, the incompatibility does not void the approach, it simply restricts the number of mechanisms directly eligible for exploration.

An ACI can be extended with the necessary signals, without affecting backward compatibility. The other data cache mechanisms will not support the new DBCP signal provided by the processor. However, because the mechanisms do not use the DBCP signal, they remain compatible with the extended interface without any modification. ACIs could become overloaded with a large number of signals. Adding more signals is not detrimental to the simulator performance and does not make the simulator code less readable, because the author of a hardware block simulator implements only the signals required. ACIs are represented as software objects. The new signals are added after inheritance of original or modified ACIs, ensuring that ACIs are not the union of all known signals but derive from each other in a tree-like fashion.

Data caches are a special form of hardware blocks, because they have a clean and clear interface with the rest of the system. Some hardware blocks, such as the commit stage of a superscalar processor, have many connections with the rest of the architecture, which change over architectures and are difficult to consider in isolation.

However, many hardware blocks are considered domains of computer architecture. These blocks have good modularity properties and can benefit from comparison. Such hardware blocks include instruction

caches, prefetchers, branch predictors, interconnects, and entire cores in a multicore processor.

There is no granularity limit to the hardware blocks that can be explored or to the corresponding interfaces. For example, within a data cache, it is possible to define a generic fine-grain interface for the connection between the data cache bank and the write buffer, which could accommodate many variations of write buffers as well as for the replacement policy and prefetch buffers (see Figures 2b, 2c, and 2d). Conversely, interfaces for coarse-grained blocks such as a whole core or an accelerator.

Although there will be variations of hardware blocks that do not fit an interface, or for which defining an interface would be, at the very least, very complex, there are numerous hardware blocks for which such interfaces can be defined and many blocks variation that will be accommodated, allowing a broad architecture exploration.

Simulator extraction and compatibility. Assuming that a hardware block is compatible with the corresponding ACI in the target server-side processor architecture, the corresponding hardware block simulator also must be compatible with the server-side processor simulator at the software level.

As mentioned earlier, the researcher can develop a custom simulator of the target hardware block. Often, this hardware block is part of an existing larger simulator. The first task is to isolate and extract the hardware block. This task is an ad hoc activity and its complexity depends on the nature of the custom simulator.

In simulators that already are modular, such as SystemC or Liberty simulators, the task of isolating a hardware block is trivial. Isolation is more complex in a monolithic simulator such as SimpleScalar.[17] For example, the SimpleScalar data cache is designed as a function called on each memory request. A modular data cache simulator executes every cycle. The data cache simulator processes all requests known at the beginning of the cycle to and from the processor and to and from the memory. As a result, the SimpleScalar data cache's mode of operation must be altered, requiring partial reprogramming.

Table 2. Data cache mechanisms.

Mechanism	Cache optimization
Content-directed data prefetching (CDP)[10]	A prefetch mechanism for pointer-based data structures
CDP + SP (CDPSP)[10]	A combination of CDP and SP
Dead-block correlating prefetcher (DBCP)[9]	Uses hit and miss patterns to drive prefetching
Global history buffer (GHB)[11]	Like stride prefetching but with varying strides
Skewed associative cache (SKEW)[12,13]	Skewing cache mapping
Stride prefetching (SP)[11]	Detects strided accesses and prefetches accordingly
Timekeeping victim cache (TKVC)[14]	Determines when a line is dead and prefetches a new one; dead lines may be sent to victim cache
Tagged prefetching (TP)[15]	Prefetches the next cache line on a miss
Victim cache (VC)[16]	A small cache to store evicted lines

After isolating a hardware block simulator, the researcher needs to privatize all its variables and methods using the C++ namespaces. Then, to achieve software-level compatibility, the researcher must wrap the simulator of the target hardware block within a module of a metasimulator. ArchExplorer uses SystemC, a modular simulation environment popular in the embedded industry, as the metasimulator environment.[18] More precisely, it uses the Unisim[3] layer on top of SystemC. Unisim provides a rigorous communication mechanism between modules, on top of SystemC signals, in the form of a three-signal hand-shaking communication protocol, derived from Liberty[2] and MicroLib.[1] This communication protocol forces to explicit, within the signals' I/O ports, how the module reacts to incoming or outgoing signals, in effect distributing the control between hardware blocks within hardware blocks themselves. If control between modules is distributed, a hardware block makes no assumption on the behavior of other hardware blocks. Therefore, a hardware block can easily be extracted and replaced with other modules or hardware blocks, which would not be the case with a central external control.

As an example, we wrapped the entire SimpleScalar simulator[17] within a SystemC module to use it as a server-side processor architecture. To plug data cache architectures

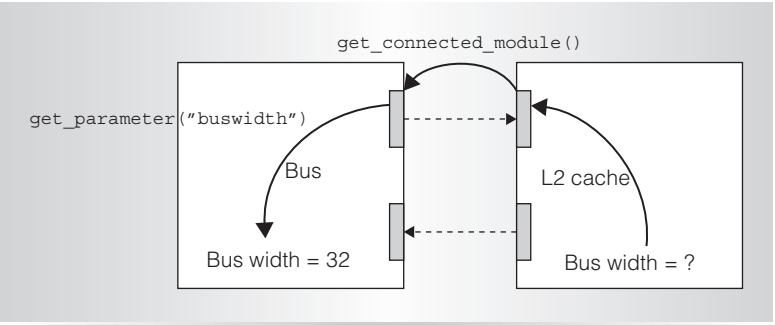

Figure 3. Self-configuration by probing connected modules. Modules have a parameter introspection interface API that lets other modules query its parameters through an interface.

into SimpleScalar, we strip it of its data caches and memory, and create explicit I/O methods to communicate with memory (caches or memory) through the module ports. Data cache modules and a synchronous DRAM (SDRAM) module to the data cache module then can be plugged, all automatically.

Beyond software module communications compatibility, running multiple hardware block simulators can be challenging if they use different models of computation[19]—for example, the order in which they process events within the same clock cycle, how they update time, and other aspects of the computations. Rather than adapt the models of computation, we could hide the model of computation in the wrapper module and require that the wrapped simulator reacts every cycle to external events. If the simulator requires a simulation engine (for example, SystemC or Liberty), we wrap the simulation engine with the simulator itself.

For example, SimpleScalar uses a pipeline loop that calls each pipeline stage in the same order, every cycle. In effect, this pipeline loop forms a simple model of computation. The loop can be broken so that it can check for external events every cycle (returning memory requests) and restart. SimpleScalar no longer is allowed to update the clock time; that role is devoted to the meta-environment.

Researchers upload their hardware block simulators with a description of valid and potentially interesting parameter ranges. Because this specification can exceed the intervals specification, we code it as a method, allowing complex parameter checking. A simulator might require parameter information from another simulator to which it is connected. Consider, for example, an architecture in which a layer-2 (L2) cache module is connected to a bus (see Figure 3). The L2 cache line must be broken into sets of the same size as the bus width. The L2 cache is the architecture block (and the simulator module) in charge of breaking down cache lines, not the bus. The L2 cache must know the bus width to break down cache lines. However, if a module is written so that it directly queries the parameters (variables) of another module, it again assumes that it knows the other module implementation, which breaks the independence properties required for automatic design-space exploration. For that purpose, a module also must have a parameter introspection interface API that allows other modules to query its parameters, preserving the independence properties of modules and simulators (see Figure 3).

Similarly, modules must implement a power and area API to obtain power and area statistics.

To wrap SimpleScalar and adapt its model of computation, we modified only 50 lines of codes. The resulting simulator has an average slowdown of 0.64 over the original SimpleScalar, factoring both the more complex data cache and synchronous DRAM. With the original data cache and SDRAM, the slowdown is 0.78, which corresponds to the wrapping overhead.

The SimpleScalar example illustrates that even if a simulator originally is monolithic and not designed to be modularized, it can be adapted with moderate effort. Not all extraction or modularization tasks will require the same effort. The effort can lighter for already modular simulators or heavier for complex monolithic simulators.

Statistical exploration of the design space

The statistical exploration of the design space is similar to genetic algorithms. Each design point corresponds to a large set of parameter values, and each parameter can be considered a gene. Genes describe modules (nature and number, for example, depth of a cache hierarchy) and each module or gene has subgenes describing their parameter

values. Genetic mutations first occur at the module level (exchanging a module for another compatible module), then at the parameter level. The database stores all gene combinations tested so far, as well as the corresponding results.

ArchExplorer uses this database to build a probability distribution of gene combinations that indicates the probability that a combination will be selected. Initially, all combinations have the same uniform probability. ArchExplorer selects a combination according to the distribution and genetically alters it at the gene and subgene levels using mutations (random modifications of modules and parameters) and crossovers (random selection of another combination and a random mix of modules and parameters). Each time a combination is simulated, ArchExplorer records the corresponding speedup (averaged over all benchmarks) and uses it to grade the combination. As a result, the distribution is biased progressively toward the best performing combinations, while genetic evolution allows the discovery of new solutions.

When a new module arrives, mutations are biased toward this new module so that it is tested rapidly (mutations normally select alternative modules and parameters uniformly). If the combinations with this module underperform known combinations, the process self-adjusts because these combinations will be selected less frequently.

We further split the distributions into area buckets corresponding to intervals of area ratio values, one distribution per bucket. An exploration usually targets a specific area budget, so exploring all possible area budgets would be inefficient. At the same time, if a combination with a smaller area than the target budget outperforms all known combinations with the target area budget, that combination should be selected; hence, the notion of buckets.

We found empirically that this approach converges quickly to good solutions. Figure 4 plots the average performance obtained for the best combination so far against the number of tested combinations.

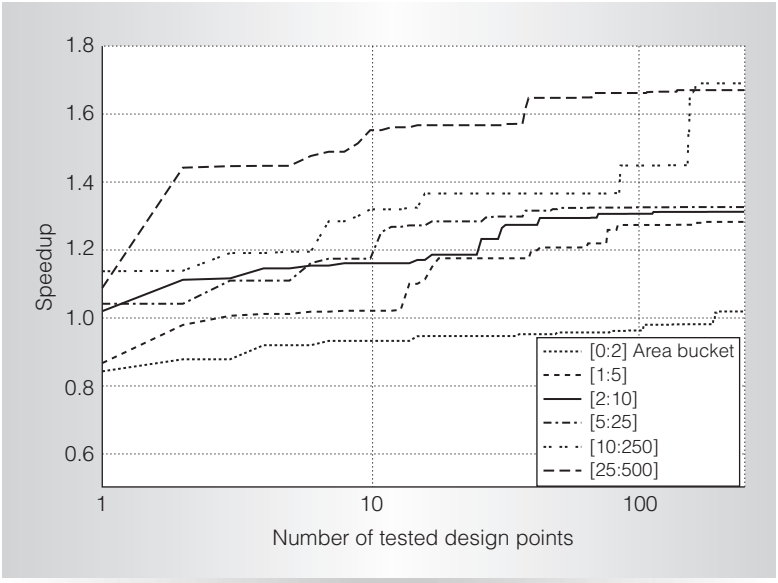

Figure 4. Convergence speed of statistical exploration. ArchExplorer achieves fast convergence through a genetic-like exploration algorithm.

Practical issues

In addition to the issues we've mentioned, the methodology used raises several unusual and practical, not technical, issues.

Recent branch prediction and data prefetching challenges promote quantitative comparison between architecture mechanisms. However, ArchExplorer features several advantages over such challenges. A researcher can compare when required rather than when the challenge event takes place. In addition, the comparison is against all accumulated mechanisms rather than against only submitted mechanisms, against any known variation of the architecture rather than only alternative but similar mechanisms. Finally, a researcher can benefit from all the accumulated exploration knowledge, which saves precious comparison and evaluation time.

At first, challenges can better assess mechanisms' relative quality by having a jury inspect the code and documentation. However, the public and permanent dissemination of the mechanisms' ranking (according to different criteria) will attract attention to the best performing mechanisms. This scrutiny provides a self-control in which high performing mechanisms can be removed if they are found to be flawed.

Public dissemination raises confidentiality issues for researchers working on new mechanisms and not yet willing to make them public, and for companies. An individual researcher can choose to upload a mechanism but keep both the exploration results and

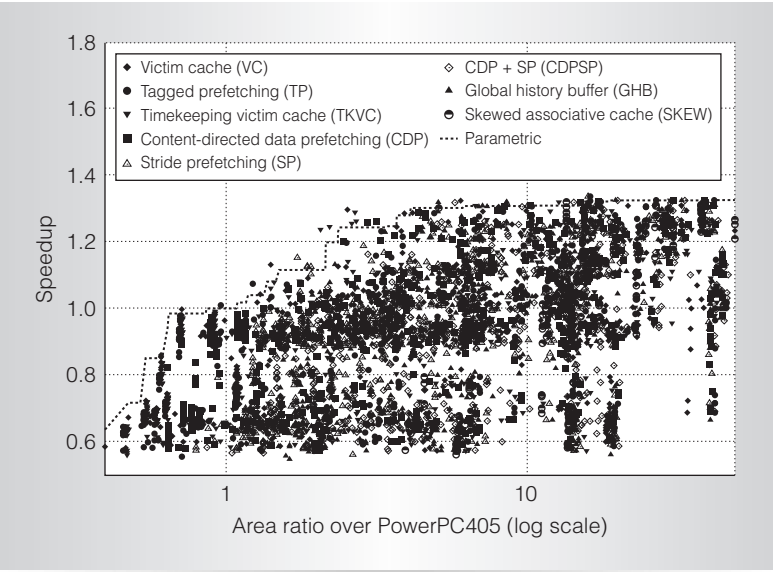

Figure 5. Parametric versus structural exploration. Advanced data cache mechanisms only moderately outperform an optimized standard data cache architecture.

simulator source private. These results do not appear in the global ranking. The researcher can make them public later.

We do not expect companies to upload their mechanisms even with the privacy option. Rather, companies will use their own target architecture and run simulations on their internal clusters. For that purpose, we provide an API to access all exploration results so a company can benefit from the knowledge accumulated in the database, as well as download the metasimulator environment and the statistical exploration tool to perform simulations internally.

Single core exploration

ArchExplorer uses an IBM PowerPC405, which is a simple 32-bit embedded RISC-processor core that includes a five-stage pipeline and 32 registers. A 70-nm version runs at the maximum frequency of 800 MHz. At 70 nm, the reference processor architecture requires 2.17 mm^2, including 0.43 mm^2 for the on-chip memory subsystem (especially data and instruction caches). The observed average memory latency is 85 cycles over all benchmarks (64 cycles for the SDRAM CAS (column address strobe) latency, best-case read memory access). When varying only the parameters of the reference architecture using the standard cache, the restricted design space contains 2,488,320 points. For the structural exploration, the various cache mechanisms from Table 2 are available, and we vary all mechanism-specific parameters, giving us more than 254 million design points.

We used the following 11 MiBench[20] embedded domain benchmarks with large input data set: bitcount, qsort, susan_e, patricia, stringsearch, blowfish_d, blowfish_e, rijndael_d, rijndael_e, adpcm_c, and adpcm_d. We compiled these benchmarks using the powerpc-405-linuxgnu-gcc cross-compiler version 4.1.0 with optimization flags-O3-static.

In addition to the metasimulation environment, ArchExplorer uses its own simulator of the PowerPC405, wrapped within a meta-environment module, together with a separate bus module and an SDRAM module. We further increase simulation speed using SimPoint, with an interval size of 10 million instructions. All modules implement the power and area API. Although we do not report power results for single-core, the power/area results follow trends similar to the performance/area results.

Data cache mechanisms versus tuned reference architecture

Figure 5 compares the performance achieved using standard data cache architectures against the performance achieved using the data cache techniques in Table 2. We distinguish the different cache mechanisms. All data cache mechanisms only moderately outperform the standard data cache architecture, when it is explored. However, these conclusions naturally depend on the target benchmarks and architectures, yet they present a rather unexpected picture of the actual benefits of sophisticated cache architectures.

Best data cache mechanisms as a function of area budget

In their data cache quantitative comparison, Gracia-Perez et al.[1] found the global history buffer (GHB) to be the best cache mechanism. Unlike their results, we vary the parameters of the reference architecture and those specific to each mechanism to assess the relative merits of these mechanisms over a broad design space (see Figure 5). Although GHB still appears to outperform competing

mechanisms for certain area sizes, almost every other mechanism emerges as the best performer for at least one area size. In fact, no mechanism clearly dominates; the best mechanism varies with the target area size.

Overall, the conclusions differ significantly from those of Gracia-Perez et al.,[1] who show that the design space must be explored broadly to truly assess architecture mechanisms' relative quantitative merits.

Convergence of exploration strategy

As Figure 4 shows, our approach quickly converges to good solutions, in which the average performance obtained for the best combination so far is plotted against the number of tested combinations.

Multicore exploration

For the multicore design-space exploration, we plugged CycleSim, a trace-driven multicore simulator, into ArchExplorer, which performed the automatic and continuous exploration. The modeled architecture was a clustered chip multiprocessor (CMP). Each cluster contained a set of cores and a last-level cache interconnected by a local bus. At the same time, all clusters are connected to main memory through a global bus. We used six parallel scientific kernels: checkSparseLU, cholesky, fft3d, kmeans, knn, and matmul. These kernels are based on direct memory access (DMA) and are highly optimized. We ported them to the Cell broadband engine variant of the StarSs[21] programming model. The traces collected from these benchmarks contained information about the required computation time for different phases in the cores as well as the intercore communications through DMA transfers.

For the multicore design space, we varied the number of clusters, the number of cores inside a cluster, the cache size, the bus bandwidth, and a core's relative performance with respect to the real machine from which we collected traces. The baseline contained four clusters of eight cores each, a total cache size of 16 Mbytes, a bus operating at 8 bytes per cycle, and a relative core performance of 1. Figure 6 shows the results of the multicore design-space exploration.

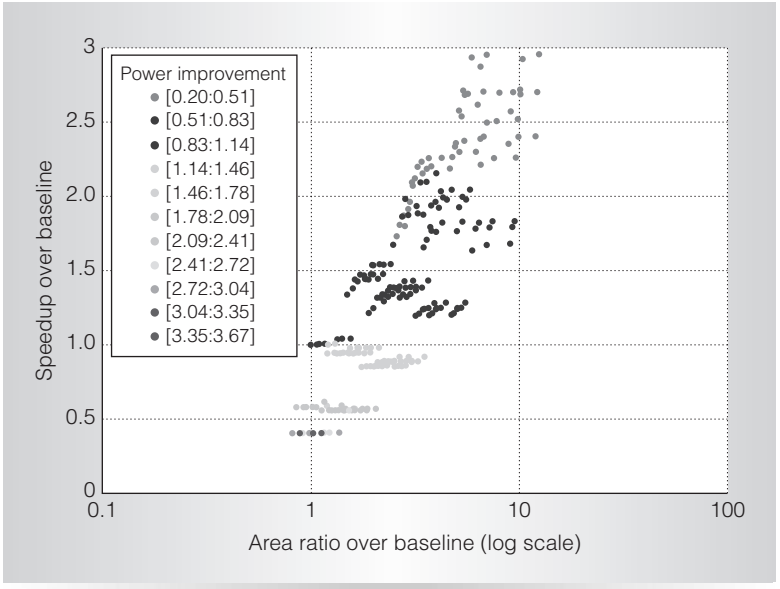

Figure 6. Multicore design-space exploration. The graph shows the average speedup for the six scientific kernels as a function of the area ratio over the baseline configuration, with the points gray-scaled according to the power improvement over the baseline.

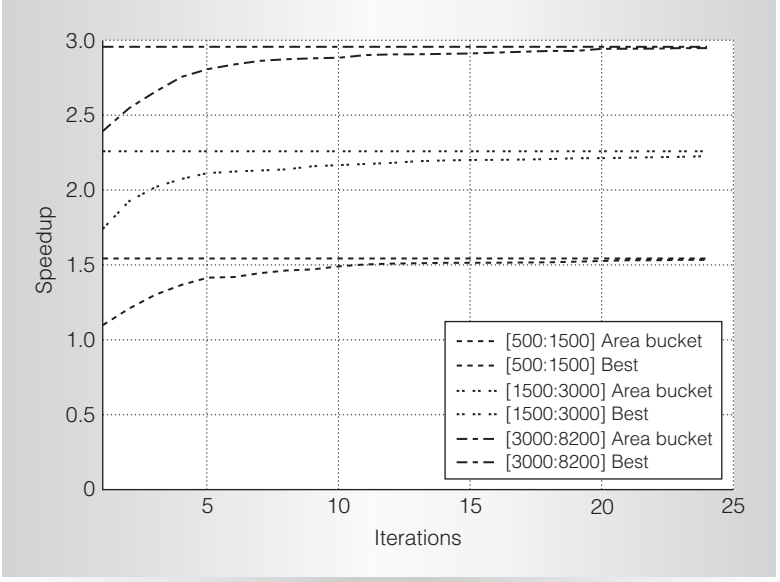

Figure 7. Convergence in the multicore design space. The multicore exploration quickly converges in 10 to 15 iterations.

Figure 7 illustrates the fast convergence of the genetic search algorithm that reaches close-to-optimal design points with 10 to 15 evaluations.

Although ArchExplorer has some limitations—not all hardware blocks, or blocks variations, can be extracted easily, plugged into a generic interface, wrapped, and uploaded—it applies to enough architecture ideas to provide a broad design space. We have implemented the website and have enlisted the data cache architectures described in this article in a permanent exploration for an embedded core. Initial results already challenge common wisdom and previous conclusions on data cache architecture research. In the future, we plan to progressively extend the exploration to more hardware blocks and more target architectures. MICRO

References

1. D. Gracia-Perez, G. Mouchard, and O. Temam, ''MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms,'' *Proc. 37th Ann. Int'l Symp. Microarchitecture,* IEEE CS Press, 2004, pp. 43-54.
2. M. Vachharajani et al., ''Microarchitectural Exploration with Liberty,'' *Proc. 35th Ann. Int'l Symp. Microarchitecture,* IEEE CS Press, 2002, pp. 271-282.
3. D.I. August et al., ''Unisim: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development,'' *Computer Architecture Letters,* vol. 6, no. 2, Sept. 2007, pp. 45-48.
4. J.S. Emer et al., ''Asim: A Performance Model Framework,'' *Computer,* vol. 35, no. 2, Feb. 2002, pp. 68-76.
5. V. Desmet, S. Girbal, and O. Temam, ''Archexplorer.org: A Methodology for Facilitating a Fair Comparison of Architecture Research Ideas,'' *Proc. 2010 IEEE Int'l Symp. Performance Analysis of Systems and Software,* IEEE CS Press, 2010, pp. 45-54.
6. N.L. Binkert et al., ''The M5 Simulator: Modeling Networked Systems,'' *IEEE Micro,* vol. 26, no. 4, July/Aug. 2006, pp. 52-60.
7. N. Hardavellas et al., ''Simflex: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture,'' *SIGMetrics Performance Evaluation Rev.,* vol. 31, no. 4, Mar. 2004, pp. 31-34.
8. M.M. Martin et al., ''Multifacet's General Execution-Driven Multiprocessor Simulator (Gems) Toolset,'' *Computer Architecture News,* vol. 33, no. 4, Sept. 2005, pp. 92-99.
9. A.-C. Lai, C. Fide, and B. Falsafi, ''Dead-Block Prediction and Dead-Block Correlating Prefetchers,'' *Proc. 28th Ann. Int'l Symp. Computer Architecture,* ACM Press, 2001, pp. 144-154.
10. R. Cooksey, S. Jourdan, and D. Grunwald, ''A Stateless, Content-Directed Data Prefetching Mechanism,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, 2002, pp. 279-290.
11. K.J. Nesbit and J.E. Smith, ''Data Cache Prefetching Using a Global History Buffer,'' *Proc. 10th Int'l Symp. High Performance Computer Architecture,* IEEE CS Press, 2004, pp. 96-105.
12. F. Bodin and A. Seznec, ''Skewed Associativity Enhances Performance Predictability,'' *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* ACM Press, 1995, pp. 265-274.
13. A. Seznec, ''A Case for Two-Way Skewed-Associative Caches,'' *Proc. 20th Ann. Int'l Symp. Computer Architecture,* ACM Press, 1993, pp. 169-178.
14. Z. Hu, M. Martonosi, and S. Kaxiras, ''Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior,'' *Proc. 29th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, 2002, pp. 209-220.
15. A.J. Smith, ''Cache Memories,'' *Computing Surveys,* vol. 14, no. 3, Sept. 1982, pp. 1473-1530.
16. N.P. Jouppi, ''Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,'' *Proc. 17th Ann. Int'l Symp. Computer Architecture,* ACM Press, 1990, pp. 364-373.
17. D. Burger, T.M. Austin, and S. Bennett, *Evaluating Future Microprocessors: The SimpleScalar Tool Set,* tech. report, Computer Sciences Dept., Univ. of Wisconsin-Madison, 1996.
18. *IEEE Std 1666TM-2005, System C Language Reference Manual,* IEEE, Mar. 2006.
19. J. Davis et al., *Overview of the Ptolemy Project,* tech. report UCB/ERL No. M99/37, Dept. of Electrical Eng. and Computer Science, Univ. of Calif., Berkeley, 1999.

20. M.R. Guthaus et al., ''Mibench: A Free, Commercially Representative Embedded Benchmark Suite,'' *Proc. IEEE 4th Ann. Workshop Workload Characterization,* IEEE CS Press, 2001, pp. 3-14.
21. P. Bellens et al., ''Cellss: A Programming Model for the Cell BE Architecture,'' *Proc. ACM/IEEE Conf. Supercomputing,* IEEE CS Press, 2006, p. 86.

Veerle Desmet is a scientific advisor at the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT). Her research interests include computer architecture, design-space exploration, multicore systems, reliability, and speculation techniques such as branch prediction, value prediction, and return address prediction. Desmet has a PhD in computer science engineering from Ghent University.

Sylvain Girbal is a research engineer at Thales TRT, France. His research interests include computer architecture simulation methodology, design-space exploration methodology, and compiler techniques. Girbal has a PhD from the University of Paris-Sud.

Alex Ramirez is an associate professor in the Computer Architecture Department at the Universitat Politècnica de Catalunya, Spain, and leader of the computer architecture group at the Barcelona Supercomputing Center. His research interests include heterogeneous multicore architectures, application specific accelerators, and architecture support for parallel programming models. Ramirez has a PhD in computer science from the Universitat Politècnica de Catalunya.

Augusto J. Vega is an engineer at the IBM Thomas J. Watson Research Center, an external collaborator at the Barcelona Supercomputing Center, and a PhD student in the Computer Architecture Department at the Universitat Politècnica de Catalunya, Spain. His research interests include performance and power modeling, simulation techniques, and architecture design of multicore processors. Vega has an MSc in computer architecture, networks, and systems from the Universitat Politècnica de Catalunya.

Olivier Temam is a senior research fellow at Inria Saclay in Paris, where he heads the Alchemy Group, and a professor at Ecole Polytechnique. His research interests include microarchitecture, simulation, compilation, programming models, and alternative computing systems. Temam has a PhD in computer science from the University of Rennes.

Direct questions or comments about this article to Olivier Temam, Inria Saclay, Batiment N, Parc Club Universite, rue Jean Rostand, 91893 Orsay Cedex, France; olivier.temam@inria.fr.