ArchExplorer.org: A Methodology for Facilitating a Fair Comparison of Research Ideas

Veerle Desmet
Ghent University, Belgium
veerle.desmet@elis.UGent.be

Sylvain Girbal
Thales TRT, France
sylvain.girbal@thalesgroup.com

Olivier Temam
INRIA Saclay, France
olivier.temam@inria.fr

Abstract

While reproducing the experimental results of research articles is standard practice in mature domains of science, such as physics or biology, it has not yet become mainstream in computer architecture. However, recent research shows that the lack of a fair and broad comparison of research ideas can be significantly detrimental to the progress, and thus the productivity, of research. At the same time, the complexity of architecture simulators and the fact that simulators are not systematically disseminated with novel ideas are largely responsible for this situation. While this methodology has a fundamental impact on research, it is by essence a practical *issue.*

In this article, we present and set up an atypical approach to overcome this practical *methodology issue, which takes the form of an open and continuous exploration through ArchExplorer, a server-side web infrastructure, that can significantly ease the process of fairly and quantitatively comparing research ideas. The web infrastructure ArchExplorer.org is now publicly open, and we demonstrate the approach with a set of data cache mechanisms. We show that this broad exploration can challenge some earlier assessments about data cache research, and even challenges the conclusions of an earlier but less thorough study on data cache comparison.*

1. Introduction

While research ideas can be appreciated purely for the new insight they provide, empirical evaluation of architecture ideas through simulation has become, rightfully or not, a fixture of any architecture research article. And new ideas are often expected to show some quantitative improvement over at least a baseline architecture, or, even better, some similar state-of-the-art mechanism(s). Many domains of sciences, such as biology or physics, often request that research articles provide the ability to reproduce experiments in order to confirm the researchers' claims; sometimes, reproducing experiments is even part of the reviewing process. However, reproducing architecture ideas of other researchers, often based on articles or on sparsely available, unmaintained or incomplete tools, is a daunting task, and it is largely comprehensible that reproducibility and systematic comparison has not become mainstream in our domain. Unfortunately, recent research by Gracia-Perez et al. [15] has shown that the lack of reproducibility and comparison can be detrimental to the progress of research. In that study, the authors have reimplemented a dozen data cache mechanisms published in the best conferences from 1982 till 2004, and compared them all on the same processor platform. They find that the best mechanism is the most recent one (proposed in 2004), and the second best is from 14 years earlier; furthermore, the progress of research between these two high points has been all but steady.

The aforementioned study highlights the risks of insufficient comparison and reproducibility, and does not propose a solution beyond modular simulation for easier reuse. While modular simulation, as proposed in SystemC [1], Liberty [36], MicroLib [15], UNISIM [4] or ASIM [12], does facilitate reuse and comparison, it assumes that all or many researchers will adopt the same simulation platform, which is only realistic within a controlled environment, like a company, as with ASIM within Intel.

The purpose of the present article is to demonstrate ArchExplorer as a *practical* approach for achieving a *fair comparison* of architecture ideas by challenging the common wisdom and even previous comparisons such as that by Gracia-Perez et al. [15] on data caches. The term *practical* is essential: what hinders better comparison/reproducibility are the time and complexity of reimplementing other researchers' work. Our methodological approach is thus focused on overcoming the practical hurdles to fair comparison.

Our approach can be summarized as an *open and permanent exploration of the architecture design space*, and it is rather atypical, as it takes the form of a web site rather than a traditional simulation environment. The approach consists of two main steps. (1) We provide a remote environment where the researcher can upload his/her own simulator, where it will be automatically run and compared against similar known mechanisms. (2) The union of all known parameters (from the baseline and the mechanisms) form a *design space* that is *continuously* explored (already over months), with the purpose of finding the best possible configurations for given area and/or power budgets. As a result, a novel mechanism is deemed profitable if it can outperform a tuned baseline configuration and all tuned competing mechanisms for the target budgets. Point (1) enables the *comparison* of architecture ideas, while point (2) enables a *fair* comparison of ideas.

After the researcher has uploaded the novel mechanism, the process of plugging in the new mechanism, simulating it and exploring the corresponding design space is entirely *automatic*. We have implemented this process and have started the permanent exploration using the data cache mechanisms surveyed in [15] plus a few others.

The results collected so far already provide a wealth of information and challenge some earlier assessments about data cache research. First, they paint a different situation than the comparison by Gracia-Perez et al. [15] led to believe. We find that there is no single dominant data cache mechanism when the area budget is varied; in fact, the best performing cache mechanism varies wildly. Second, in many cases (area budgets), most data cache mechanisms turn out to offer little performance advantage over a standard cache once the standard cache design space is duly explored. The first point shows that it is necessary to have mechanisms to compare with available at all time as the relative merits of each mechanism can change with each new target architecture (e.g., area budget). The second point shows that a fair comparison, where the baseline architecture and all other known alternative mechanisms are configured in the best possible way (for the given area budget), can offer a strikingly different view of the merits of a new mechanism.

Naturally, our methodological approach raises several issues and has some limitations, which we discuss in the following sections, especially the effort required to make an existing simulator compatible with this open exploration process, the architecture features/ideas which are eligible to this methodological approach, and the way companies can benefit from such an open process.

Beyond facilitating comparison, our automatic and permanent exploration may also assist a designer by proposing best overall architectures for given budgets based on explored parameters and all known alternative mechanisms so far. This possibly leads to solutions which can complement the ones derived from intuition and experience.

2. A Comparison Methodology Relying on Open and Continuous Exploration

The ArchExplorer approach is summarized in Figure 1. In short, a researcher adapts his/her custom simulator to make it compatible with our environment, uploads the simulator together with a specification of valid and potentially interesting parameter ranges, and the mechanism is immediately added to the continuously explored design space. The architecture design points are statistically selected/explored, and for each architecture design point, the compiler is automatically retuned for a truly meaningful comparison with other architecture design points, and with the benchmarks being recompiled accordingly. After the set of benchmarks

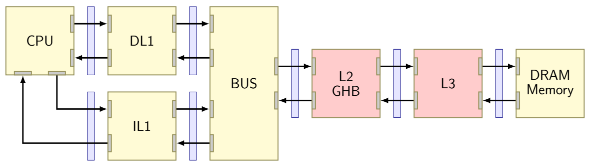

Figure 2. *Composing architectures.*

has been simulated for this design point, performance results are accumulated in the database, and the ranking of the mechanisms is automatically updated and publicly disseminated on the web site. The process is described in more details and discussed below.

2.1 Automatically composing architectures

As mentioned above, the first step consists of adapting the custom simulator of the researcher to our environment. On one hand, we do not require researchers to use a given common simulation platform since that would not be realistic: in many cases, researchers have invested a lot of effort in building or becoming familiar with a simulation platform, and it is unlikely he/she would discard that effort, even for the sake of comparison. On the other hand, there is no free lunch: we do not pretend that one can integrate a simulator part within another simulation environment without any modification. However the task often appears far more daunting than what it really is in practice. Moreover, modular simulation environments like ASIM, Liberty or SystemC, or modular simulators like M5 [6], SimFlex [18] or GEMS [28] are progressively making users more sensitive to the productivity benefits of modularity, which, in turn, will make simulators even more easily compatible with our approach.

Architecture Communications Interfaces. The first step is to ensure that both architecture parts, the uploaded hardware block and the server-side architecture (e.g., an uploaded data cache and a server-side processor), are compatible at the *hardware* level; for that purpose, they should agree on a set of input and output control and data signals. This set of signals forms a communication interface which we term *Architecture Communications Interfaces*, as an analogy to software-level APIs (Application Programming Interfaces). For instance, the processor/memory interface as depicted in Table 2, enables processors to connect with a large range of cache mechanisms, and allows them to compose arbitrarily deep cache hierarchies with interface-abiding caches, see Figure 2.

ACIs raise two main questions. (1) Do we need to design a new ACI for each new hardware block variation? (2) Which hardware blocks, typically studied in architecture research, are eligible to an ACI definition?

Extending ACIs to accommodate new mechanisms. For most data cache mechanisms, the innovations proposed are *internal* to the block and these innovations have little or no impact on the interface with the processor and the memory in many, though not all, cases. Of the 9 data cache mecha-

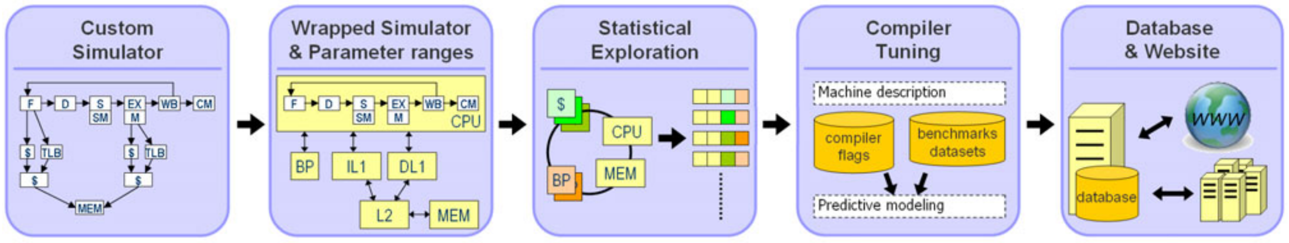

Figure 1. *Overview.*

Acronym	Cache Optimization
VC	Victim Cache [23] A small cache to store evicted lines
TKVC	Timekeeping Victim Cache [20] Determine when a line is dead and prefetch a new one, dead lines may be sent to victim cache
TP	Tagged Prefetching [34] Prefetch next cache line on a miss
SP	Stride Prefetching [5] Detect strided accesses and prefetch accordingly
CDP	Content-Directed Data Prefetching [8] A prefetch mechanism for pointer-based data structures
CDPSP	CDP + SP [8] A combination of CDP and SP
DBCP	Dead-Block Correlating Prefetcher [25] Use hit and miss patterns to drive prefetching
GHB	Global History Buffer [30] Like stride prefetching but with varying strides
SKEW	Skewed Associative Cache [33, 7] Skewing cache mapping

Table 1. *Data cache mechanisms.*

`address`	bi-directional, 32 bits Memory request address.
`data`	bi-directional, path width Data for read and write requests.
`size`	bi-directional, $log_2(max(\#bytes))$ bits Request size in bytes.
`command`	processor → cache, 3 bits
proc./L1	Request type (`read`, `write`, `evict`, `prefetch`).
L1/L2	Request type (`read`, `write`, `evict`, `prefetch`, `readx`, `flush`).
proc./mem.	Request type (`read`, `write`, `evict`, `prefetch`, `readx`, `flush`).
`cachable`	processor → cache, 1 bit Whether or not the requested address is cachable.

Table 2. *Processor/Memory interface.*

nisms listed in Table 1, all mechanisms but one, DBCP [25], are compatible with the interface of Table 2, which itself corresponds to the standard data cache interface. Moreover, an ACI can be extended with the necessary signals, without affecting backward compatibility.

Hardware blocks eligible to ACIs. It can also be argued that 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, effectively have a large set of connections with the rest of the architecture, which change over architectures, and are thus difficult to consider in isolation. However, there are quite a few hardware blocks considered as "domains" of computer architecture, which have good modularity properties and which would thus benefit from fair comparison. A non exhaustive list of such hardware blocks includes: instruction caches, TLBs, prefetchers, branch predictors, interconnects (buses and network on chips) and network topology, main memory (DRAMs, SRAMs, RDRAM,...), any co-processor in a heterogeneous system (popular in embedded SoC or upcoming general-purpose heterogeneous multi-cores), cores themselves in multi-cores, functional units,...

Simulator extraction and compatibility. Besides hardware-level compatibility enforced through ACI, the simulator must naturally be compatible at the *software* level.

Isolating and extracting the hardware block simulator. As mentioned before, we let the researcher develop his/her own custom simulator of the target hardware block. Often, this hardware block will be part of an existing larger simulator, and the first task is to isolate and extract the hardware block. This is an ad hoc task, and its complexity depends on the nature of the custom simulator.

Wrapping and distributing control. Once a hardware block simulator has been isolated, we then privatize all its variables and methods simply using the C++ `namespaces`. Then, in order to achieve software-level compatibility, we request that the researcher *wraps* the simulator of the target hardware block within a module of a *meta-simulator*. We use SystemC, an existing modular simulation environment, popular in embedded industry (it is an IEEE standard [2]), as the meta-simulator. More precisely, we exactly use the UNISIM [4] layer on top of SystemC. UNISIM provides a rigorous communication mechanism between modules, on top of SystemC signals, in the form of a 3-signal hand-shaking communication protocol, derived from Liberty [36] and MicroLib [15]. This communication protocol forces to

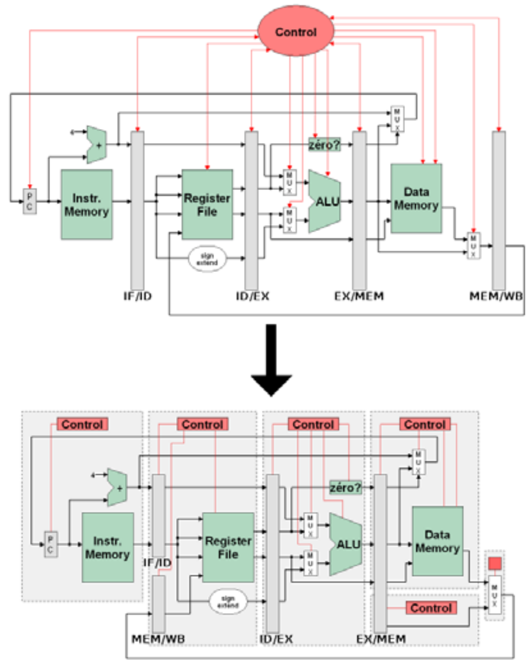

Figure 3. *Central vs. distributed control.*

explicit module behavior upon incoming/outgoing communications at the level of input/output ports, in effect *distributing the control* between hardware blocks, see Figure 3 for a conceptual view of central versus distributed control. The benefit of distributed control is that a hardware block makes *no assumption* on the behavior of other hardware blocks, and therefore, it can be easily extracted and replaced with other modules/hardware blocks. For instance, we wrapped SimpleScalar in one module after modifying a few tens of lines in order to break the control loop to memory.

Models of computation. Beyond software module communications compatibility, running multiple different hardware blocks simulators can be challenging if they use different *models of computation* [10], i.e., the order in which they process events within the same clock cycle, how they update time, and so on. Rather than adapting the models of computation, we take a more simple, pragmatic approach: we simply hide the model of computation within the wrapper module, and impose that the wrapped simulator can react every cycle to external events.

Power/energy/area behavior. Our environment also includes "services. These are software APIs that provide a standardized interface to energy and power models (as well as debugging, loader and other services). The power and area API that any module must implement in order to obtain power, energy and area statistics. Any module supporting this API (a few calls per module) will have access to the models. Moreover, the power model can be swapped without changing the simulator modules. Currently, we use CACTI as power model.

Confidentiality issues. The open web-based infrastructure raises confidentiality issues for both a researcher or a company. For researchers uploading a novel mechanism, we provide the option of simply hiding results, they are not listed on the public comparison pages. For companies, we provide an API and the exploration engine so that the exploration can be performed on company servers while still benefiting from the exploration knowledge gathered in the public database.

2.2 Automatically tuning the compiler for the architecture design point

The potential impact of the compiler on architecture design decisions is often overlooked. Usually, two architecture design points P_1 and P_2 are compared using benchmarks compiled with a compiler tuned for the baseline architecture. In reality, programs will be run with a compiler tuned for the final architecture. So while P_1 may perform better than P_2 using the baseline-tuned compiler, that relative performance could be *reversed* if the compiler is first tuned to each design point and the benchmarks recompiled accordingly.

This is all the more true since the performance benefits of compiler optimizations are similar, sometimes even higher, than the performance benefits of architecture optimizations. In order to illustrate that point, we attempted to squeeze the maximum "software optimization" performance out of every benchmark by tuning the compiler for each benchmark individually, for the standard architecture; this tuning is achieved by statistically exploring a set of compiler optimizations combinations. We find that the potential performance improvement due to software tuning is on par with the usual performance improvement due to architecture improvements, such as enhanced data caches for instance, see Figure 4. Later on, in Section 4, we will further show that a significant fraction of comparisons between two architecture designs points can be reversed if the compiler is tuned for each design point rather than for the baseline.

As a result, for each architecture design point, we first tune the compiler by statistically exploring combinations of compiler optimizations. While manual compiler tuning is a tedious process, recent research in iterative compilation [9] has shown that it is possible to automatically tune a compiler for a given architecture platform. Moreover, this process was shown to converge rapidly because the density of good compiler optimizations combinations is usually high [3]. Still, compiler exploration further multiplies the size of the design space. In order to reduce the impact on the design space size, we pre-select a number of optimizations combinations found to perform well for at least one benchmark on the standard architecture, and use that set of combinations as a high-quality compiler tuning sub-space. The corresponding optimizations are listed in Table 4.

2.3 Statistical exploration of the design space

Our statistical exploration of the design space is similar though not exactly like genetic algorithms. The principle is

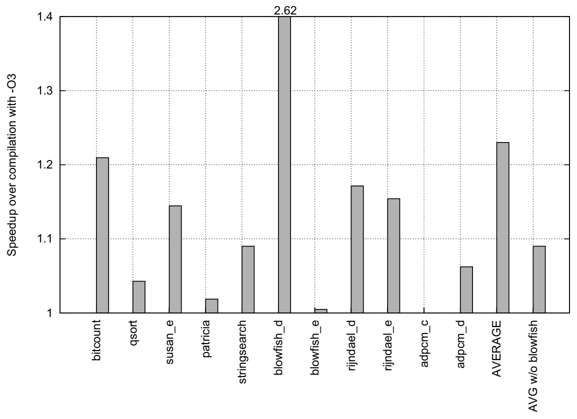

Figure 4. *Speedup after tuning the compiler on the reference architecture.*

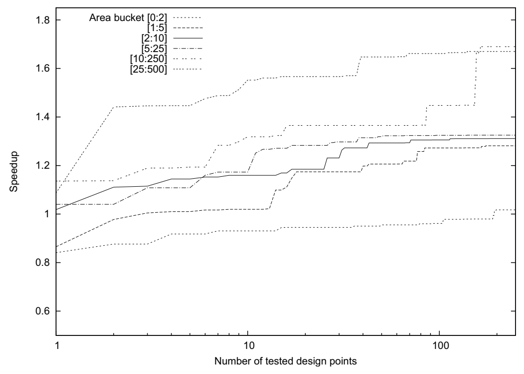

Figure 5. *Convergence speed of statistical exploration.*

that each design point corresponds to a large set of parameter values, and each parameter can be considered as a gene. We in fact distinguish two gene levels: genes describing modules (nature and number, e.g., depth of a cache hierarchy), and for each module/gene, the sub-genes describing modules parameters values. The genetic mutations first occur at the module level (swapping a module for another compatible one), and then at the parameter level. The database stores all gene combinations tested so far, and the corresponding results.

This database/knowledge is used to build a probability distribution of genes combinations which indicates the probability that a combination should be selected. Initially, all combinations have the same uniform probability of being selected. A combination is selected according to the distribution and it is genetically altered at the gene and sub-gene level using mutations (random modifications of modules and parameters) and cross-overs (random selection of another combination, and random mix of modules and parameters). Each time a combination is simulated, the corresponding speedup (averaged over all benchmarks) is recorded and used to grade the combination. As a result, the distribution is progressively biased towards the best performing combinations, while genetic evolution allows to discover new solutions.

When a new module comes in, we initially bias mutations towards this new module so that it is rapidly tested (mutations normally uniformly select alternative modules/parameters). If the combinations with this module underperform known combinations so far, the process self-adjusts as these combinations will become more rarely selected.

We further split the distributions into *area buckets* corresponding to intervals of area ratio values, one distribution per bucket. Indeed, an exploration usually targets a specific area budget, so it would not make sense, and it would be inefficient, to explore all possible area budgets. At the same time, if a combination with a smaller area than the target budget outperforms all known combinations with the target area budget, it should be selected; hence, the notion of buckets.

We empirically found that our approach quickly converges to good solutions as shown in Figure 5, where the average performance obtained for the best combination so far is plotted against the number of tested combinations.

3. Experimental Framework

Architecture. We use an IBM PowerPC405 [21], which is a simple 32-bit embedded RISC-processor core including a 5-stage pipeline and 32 registers. The core has no floating-point unit. We consider a 70nm version running at the maximum frequency of 800MHz (to date, the PowerPC405 has been taped out at 90nm with a maximum clock frequency of 667MHz). At 70nm, the reference processor architecture requires 2.17mm2, including 0.43mm2 for the on-chip memory sub-system (especially data and instruction caches); the observed average memory latency is 85 cycles over all benchmarks (64 cycles for the SDRAM CAS latency, best case read memory access).

The on-chip part of the memory sub-system of our reference architecture is described in Table 3. The baseline parameters of this reference architecture are shown in Table 3. The modules database contains a standard cache and the cache mechanisms of Table 1 except for DBCP. The L1 cache access times are computed using CACTI [35].

When varying only the *parameters* of the reference architecture (parametric exploration using the standard cache), our restricted design space contains 2,488,320 points. With the cache mechanisms, it increases to more than 254 million design points. Since we allow cache hierarchies of up to 4 levels, the full design space contains nearly 10^{24} points.

Benchmarks and compiler. We use the MiBench [16] embedded domain benchmarks listed in Table 4. We only consider 11 out of the 29 MiBench. 4 were excluded because of long running time (the exploration is naturally computing re-

Parameter description		Range	Reference
ppc	IL1CacheLines	64,128,256	256
ppc	IL1Associativity	1,2,4,8	2
bus	BufferSize	3,10,30,100	30
cache	LineSize	32,64,128	32
cache	CacheLines	even power of 2 min 256, max 65536	256
cache	Associativity	1,2,4,FA	2
cache	MSHR	1,8,16,64	8
cache	MSHRRead	1,8,64,256	8
dram	Banks	2,4	4
dram	Rows	2048,4096,8192	2048
dram	Cols	256,512,1024	256
dram	CtrlQueueSize	8,16,32	16
CDP	PrefecthDepthThreshold	3,8,15	3
CDP	CDPEntries	32,128,512	128
CDPSP	(*see CDP and SP*)		
GHB	GHBITEntries	256,512	512
GHB	GHBEntries	256,512	512
GHB	GHBPCShift	1,2,3,4	3
GHB	GHBDepth	1,2	1
SKEW	mappingfunction	XOR-ing	XOR-ing
SP	SPEntries	256,512,1024	512
SP	SPPCShift	3,5,8	3
TKVC	TKRefresh	256,512,1024	512
TKVC	TKTop	1,4,16	4
TKVC	TKThreshold	1,4	1
TKVC	TKVCCacheLines	4,16,64,256	16
TKVC	TKVCAssociativity	1,2,4,FA	1
TP	PrefecthQueue	4,8,16,32	16
VC	VCCacheLines	4,16,64,256	16
VC	VCAssociativity	1,2,4,FA	1

Table 3. *Design parameters ranges.*

Benchmark	Best compiler flags
bitcount	-O1 -falign-jumps=36 -fschedule-insns2 -fsingle-precision-constant
qsort	-O3 -falign-functions=56 -falign-jumps=51
susan_e	-O1 -fno-ivopts -fno-tree-fre
patricia	-O0 -fsched-stalled-insns-dep=34 -fsched-stalled-insns=62 -falign-loops=63
stringsearch	-O1 -fno-rerun-cse-after-loop -fschedule-insns2
blowfish_d	-O1 -fno-rerun-cse-after-loop -fschedule-insns2
blowfish_e	-O1 -falign-functions=37 -falign-jumps=8 -falign-loops=40
rijndael_d	-O1 -falign-functions=30 -falign-loops=60 -fregmove
rijndael_e	-O1 -fsched-stalled-insns-dep=5 -fgcse-lm -finline-functions-ftree-pre
adpcm_c	-O3 -fsched-stalled-insns-dep=54 -falign-loops=47
adpcm_d	-O3 -fsched-stalled-insns-dep=5 -fno-tree-lrs

Table 4. *Benchmarks, and best compiler flags.*

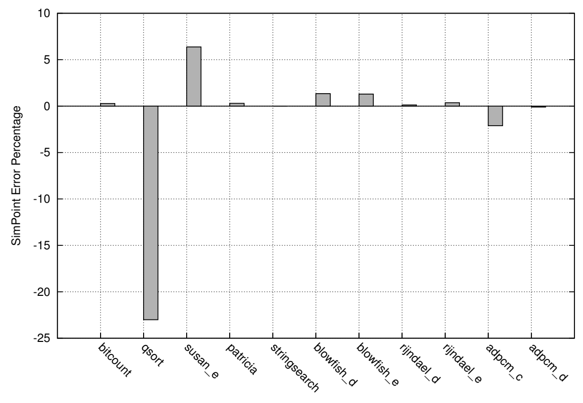

Figure 6. *SimPoint error (in %) on reference architecture.*

source intensive), 7 programs had cross-compilation or endianness issues (e.g., writing several bytes to be used as a 32-bit integer), and 7 (including the 4 `tiff` benchmarks) because our infrastructure could not correctly checkpoint them for later use with SimPoint (see below). We use the large input data set of the MiBench.

They are compiled using the *powerpc-405-linux-gnu-gcc* cross-compiler version 4.1.0; the default optimization flags are `-O3 -static`. We restrict the compiler design space by pre-selecting a number of compiler flags combinations. The best flags found for each benchmark, during the compiler exploration process on the reference architecture, are listed in Table 4. This list of optimizations combinations forms a small but high-quality compiler exploration space which was used during the automatic joint hardware/compiler exploration.

Simulation. Besides the aforementioned meta simulation environment, we use our own simulator of the PowerPC405, wrapped within a meta environment module, together with a separate bus module and an SDRAM module.

In order to speed up design space exploration, we use SimPoint [17] with an interval size of 10 million instructions. We configure SimPoint to select up to 30 fixed-length simpoints with a coverage percentage of 98%. Measurements are done for each of the simpoints, and the results are aggregated per benchmark according to the weights provided by SimPoint. All speedups are computed against the reference architecture. In order to account for the performance noise introduced by SimPoint, the simulation error of each benchmark on the reference architecture is shown on Figure 6. The error is computed against full runs; 3 out of 15 used benchmarks could not be validated because of excessively long full-run simulation time. `quicksort` exhibits a significant error of 23%, 6.5% for `susan_e`, a smaller error of 2% for `adpcmc` and around 1% or less for all other benchmarks. `susan_e` and `adpcm_c` have a small enough execution time that we can resort to full runs for these benchmarks, however we exclude `quicksort` from all statistics reported in this article.

Note that SimPoint is a program-dependent approach. So, for each compiler optimization tested during the compiler exploration, and for each benchmark, a functional simulation of the full run is necessary to extract the corresponding basic block vectors (BBVs) and find the simpoints.

Finally, all modules implement a power and area API. The power and cost of the different cache mechanisms are modeled with CACTI. And we similarly adapted CACTI 4.1 [35] so as to receive requests through that API. The main power modeling approximation lays in the cost of the prefetch buffers, which we model as SRAM tables, since CACTI cannot yet model such FIFOs; however, these buffers are usually much smaller than the caches themselves, and therefore do not significantly bias the cost estimates.

4. Combining Quantitative Comparison and Exploration

We now demonstrate the whole ArchExplorer process to the exploration of a processor memory sub-system. We want to compare the behavior of all aforementioned data cache architectures, composed into many possible hierarchies, for a large set of area budgets. All charts in this paper are averages over the benchmark set, we refer to the ArchExplorer.org website for individual benchmark performance.

4.1 Data cache mechanisms versus tuned reference architecture

In Figure 7, we compare the best performance achieved using standard data cache architectures against the performance achieved using the data cache techniques of Table 1. In the former case, we only vary the typical processor parameters, see Table 3, hence the term *parametric* exploration, and the curve represents the best performance achieved for each design area size. In the latter case, we vary the data cache structure and name this *structural* exploration. In Figure 7, we only plot the best structural points above the parametric envelope, and distinguish between the different L1 mechanisms.

We find that all data cache mechanisms only moderately outperform the standard data cache architecture, when it is duly explored. In some studies, the overhead area of the proposed mechanism is offset in a simple way against the standard data cache, for instance by increasing its size. However, to our knowledge, there is rarely a full exploration of the data cache architecture to find the best possible configuration for the target area budget. As it turns out, it seems difficult to outperform the standard data cache architecture when the additional area is properly used. However, as a precaution, we want to point out that these conclusions can evolve as the exploration finds new design points, we already observed that the overall profile changed over time. Moreover, these conclusions are specific to the current benchmarks and processor architectures, and will likely evolve as more benchmarks and architectures are considered.

4.2 Best data cache mechanisms as a function of area budget

In the data cache quantitative comparison by Gracia-Perez et al. [15], GHB was found to be the best cache mechanism. Unlike in [15], in Figure 7, we have varied the parameters of the reference architecture and the parameters specific to each mechanism in order to assess the relative merits of these mechanisms over a broad design space. While GHB still appears to outperform competing mechanisms for certain area sizes, almost every other mechanism also emerges as the winner for at least one area size. And in fact, there is no clearly dominant mechanism, the best mechanism varies wildly with the target area size. For instance, SKEW and CDPSP perform better for large area budgets, TKVC works well for small budgets, and VC performs well across the spectrum. Overall, the conclusions are quite different from [15], which shows that the design space must be broadly explored to truly assess the relative quantitative merits of architecture mechanisms. However, again, these conclusions can evolve as the exploration progresses, and more benchmarks and architectures are considered.

4.3 Hierarchical compositions of hardware blocks

The exploration is not limited to replacing a standard data cache architecture with one of the data cache mechanisms. These mechanisms can be combined together *automatically* because their interface to memory is the standard processor/memory interface defined in Section 2; as a result, on the memory side, it is possible to either plug a bus or another cache. Some properties cannot be investigated automatically though, such as non-inclusive caches, and they are not included in our exploration. Extending the exploration to hierarchies yields the result of Figure 8. As expected, the maximum performance can increase, though only for large area budgets. However, combining data cache mechanisms does not yield significantly better performance than combining standard data caches either, except for a few cases. The performance bump of the parametric envelope, around an area ratio of 40, denotes that, for the parameter ranges in Table 3, below that area size, there always exists a standard cache parametrization which outperforms a data cache hierarchy.

4.4 Multi-Criteria exploration

Figure 9 is the same as Figure 8, except that the energy ratio over the baseline architecture (energy improvement) is shown using gray shades: the mapping between the gray shades and the energy improvement is shown in the figure legend. The conclusions drawn for speedup generally hold for energy improvement: few mechanisms outperform a properly tuned baseline architecture, and most outperforming mechanisms exhibit similar energy behavior, meaning there is again no outstanding mechanism either, over a large area domain, when it comes to energy improvement.

4.5 Hardware-Only exploration vs. joint hardware/software exploration

In order to assess the sensitivity of the design process to compiler tuning, we have considered that the different compiler optimizations combinations of Table 4 form a restricted software design space, and we have tried each such combination on all hardware combinations. Therefore, for each hardware configuration, there is now a *base* compiler performance, obtained with the default compiler optimizations, and a *best* compiler performance, which is the best among all possible compiler optimizations combinations.

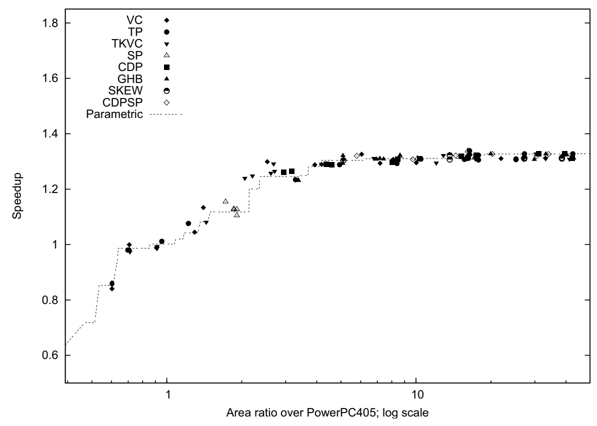

Figure 7. *Parametric vs. structural exploration.*

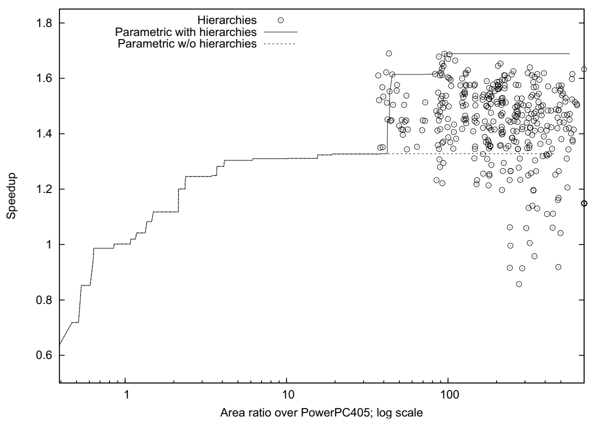

Figure 8. *Hierarchical compositions of data cache mechanisms.*

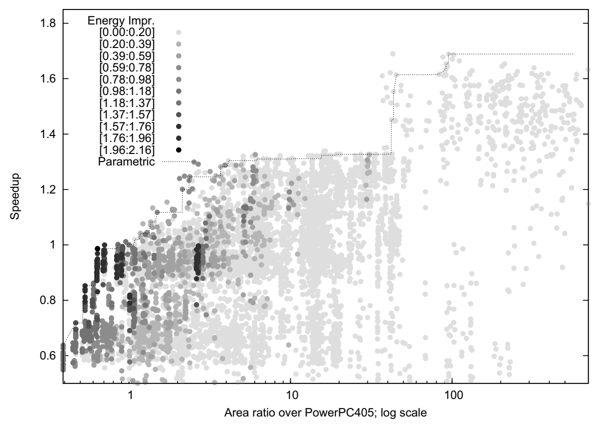

Figure 9. *Speedup and energy improvements.*

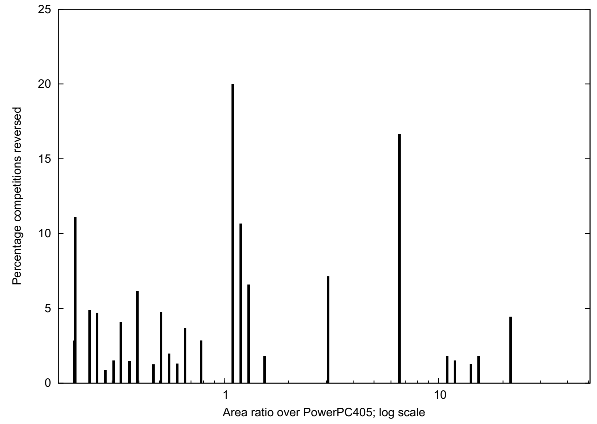

Figure 10. *Fraction of hardware design decisions reversed by software exploration.*

Then, we emulate a "competition" between all hardware configurations: the performance (speedups) of any pair of hardware configurations (hc_1, hc_2) are compared. If $hc_1 > hc_2$ for the base software configuration, but $hc_1 < hc_2$ for the best software configuration (or vice-versa), then software exploration has reversed the design decision. In Figure 10, we report the fraction of all competitions which have been reversed per area interval (divided into area buckets as for Figure 10). The fraction of reversed decisions can reach up to 20%, so that the lack of compiler tuning during the design process may indeed significantly affect design decisions. Moreover, this fraction is most likely a conservative estimate considering the restricted size of our compiler design space. Still, we note for now that the fraction of reversed decisions varies wildly across area budgets. As we accumulate more statistics over time, we will be able to determine if this randomness is a methodological artifact due to a yet insufficiently exhaustive exploration of the compiler+architecture design points, or if there is effectively a chaotic impact of compiler tuning on design decisions.

5. Related work

Literature provides many examples of targeted attempts at design-space exploration. For instance, Hsu [19] explores the cache design space for large-scale CMPs. Emer [13] has shown that it is possible to create efficient branch predictors by decomposing branch prediction algorithms into elementary primitives and then composing them, thereby creating new branch predictors. Monchiero [29] has parametrized a shared-memory multi-processor to explore the effects of the number of cores, L2 cache size, and processor complexity. Li [27] performs a similar study but also leverages operating voltage and clock frequency.

Several research works attempt to generalize DSE and provide frameworks for architecture exploration, though they still do not bring design-space exploration beyond parameter exploration. Magellan [24] is a framework for multi-core exploration, which embeds power/area measurement, statistical exploration techniques and exposes a large range of multi-core parameters. Palermo [31] focuses on the exploration of embedded systems, particularly heuristics to rapidly converge to pareto-optimal configurations (performance, power, delay). Similarly, Pimentel [32] proposes the Sesame framework for design-space exploration in the context of system-on-chips; it uses multiple abstraction levels in order to speed up the exploration.

For both architectural and compiler exploration, there is a large and recent breadth of works on building statistical models using machine-learning techniques which show that it is possible to rapidly converge to good solutions amid a huge design space. While we resorted to simple versions of such techniques in our study, we contemplate applying several of the techniques below for further improving convergence speed. Ipek [22] shows that it is possible to train a neural network so that it accurately emulates a processor simulator, thereby allowing very fast design-space exploration. More recently, Dubach [11] extended Ipek's work to design program-independent models for the same purpose. Lee [26] uses regression modeling for similarly exploring joint microarchitecture/application design space. Finally, adaptive and iterative compilations have demonstrated compilers can be rapidly retargeted to a new architecture platform, again using similar machine-learning techniques [3, 14].

6. Conclusions and Future Work

Even if it is natural for researchers to prefer focusing on finding new ideas rather than addressing gruesome methodological issues, there is empirical evidence that the lack of fair comparison can be profoundly detrimental to the progress of research; as a result, methodological issues should receive much more attention. We present through ArchExplorer.org an atypical but practical methodological approach for addressing the lack of fair quantitative comparison of architecture research ideas. Our approach consists of an open and continuous exploration of the architecture design space, where any researcher can propose/upload a new mechanism to explore and compare. Our methodology especially aims at overcoming the practical difficulties of reimplementing and fairly comparing against a large set of alternative mechanisms.

Our approach makes the following contributions. (1) The exploration knowledge accumulated in the database avoids the repeated tuning of the baseline and all other known mechanisms, which thus considerably speeds up any comparison, and provides the best possible comparison points for given area and/or power budgets. (2) Comparison fairness will only increase with the number of submitted mechanisms, as the approach gets traction. (3) The joint architecture/compiler exploration improves the accuracy and fairness of architecture design points comparisons. (4) In addition to statistical exploration and sampling, the continuous exploration over a long period of time is a practical means for exploring a huge design space. (5) The process is open, with ranking publicly displayed and frequently updated, while proposing several privacy or remote access options for addressing confidentiality concerns.

Even though the process has some limitations (not all hardware blocks, or blocks variations, can be easily extracted, plugged into a generic interface, wrapped and uploaded), it applies to enough architecture ideas to provide an already very broad design space. We have already opened the web site and enlisted the aforementioned data cache architectures into a permanent exploration for an embedded core. The first 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.

Acknowledgments

We would like to thank Sami Yehia for his helpful comments. This work is partially supported by Ghent University, HiPEAC, and the European SARC project No. 27648.

References

[1] SystemC, OSC Initiative. Technical report, 2003.

[2] *IEEE Computer Society, IEEE Std 1666^{TM}-2005, System C Language Reference Manual*. IEEE 3 Park Avenue, New York, NY 10016-5997, USA, Mar. 2006.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*, pages 295–305, Mar. 2006.

[4] D. I. August, J. Chang, S. Girbal, D. Gracia Pérez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. UNISIM: An open simulation environment and library for complex architecture design and collaborative development. *Computer Architecture Letters*, 6(2):45–48, Sept. 2007.

[5] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the International Conference on Supercomputing*, pages 176–186, 1991.

[6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July 2006.

[7] F. Bodin and A. Seznec. Skewed associativity enhances performance predictability. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 265–274, June 1995.

[8] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–290, Oct. 2002.

[9] K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1):7–22, 2002.

[10] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuerdorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the Ptolemy project. Technical Report UCB/ERL No. M99/37, EECS, University of California at Berkeley, 1999.

[11] C. Dubach, T. M. Jones, and M. F. P. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *Proceedings of the 40th Annual International Symposium on Microarchitecture*, pages 262–273, Dec. 2007.

[12] J. S. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. L. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, Feb. 2002.

[13] J. S. Emer and N. C. Gloy. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 304–314, June 1997.

[14] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Languages, Compilers, and Tools for Embedded Systems*, pages 78–86, June 2005.

[15] D. Gracia Pérez, G. Mouchard, and O. Temam. MicroLib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 43–54, Dec. 2004.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14, Dec. 2001.

[17] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and more flexible program analysis. In *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, June 2005.

[18] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):31–34, Mar. 2004.

[19] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. Exploring the cache design space for large scale CMPs. *SIGARCH Computer Architecture News*, 33(4):24–33, Nov. 2005.

[20] Z. Hu, M. Martonosi, and S. Kaxiras. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 209–220, May 2002.

[21] IBM. PowerPC 405 CPU Core. Sept. 2006.

[22] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 195–206, Oct. 2006.

[23] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, June 1990.

[24] S. Kang and R. Kumar. Magellan: A search and machine learning-based framework for fast multi-core design space exploration and optimization. In *Proceedings of the Design, Automation, and Test in Europe*, pages 1432–1437, Mar. 2008.

[25] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 144–154, June 2001.

[26] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 185–194, Oct. 2006.

[27] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP design space exploration subject to physical constraints. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pages 15–26, Feb. 2006.

[28] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.

[29] M. Monchiero, R. Canal, and A. González. Design space exploration for multicore architectures: a power/performance/thermal view. In *Proceedings of the 20th annual International Conference on Supercomputing*, pages 177–186, June 2006.

[30] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 96–105, Feb. 2004.

[31] G. Palermo, C. Silvano, and V. Zaccaria. Multi-objective design space exploration of embedded systems. *J. Embedded Comput.*, 1(3):305–316, 2005.

[32] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, Feb. 2006.

[33] A. Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, May 1993.

[34] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):1473–530, Sept. 1982.

[35] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.0 technical report. Technical report, HP Labs, 2006.

[36] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 271–282, Nov. 2002.