MicroLib: A Case for the Quantitative Comparison of Micro-ArchitectureMechanisms

Daniel Gracia Perez, Gilles Mouchard, Olivier Temam

▶ **To cite this version:**

Daniel Gracia Perez, Gilles Mouchard, Olivier Temam. MicroLib: A Case for the Quantitative Comparison of Micro-ArchitectureMechanisms. Workshop on Duplicating, Deconstructing, and Debunking, Jun 2004, Munich, Germany. inria-00001109

HAL Id: inria-00001109
https://inria.hal.science/inria-00001109

Submitted on 9 Feb 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms

Daniel Gracia Pérez   Gilles Mouchard
Olivier Temam

LRI, Paris Sud/11 University   INRIA Futurs, France

Abstract

While most research papers on computer architectures include some performance measurements, these performance numbers tend to be distrusted. Up to the point that, after so many research articles on data cache architectures, for instance, few researchers have a clear view of what are the best data cache mechanisms. To illustrate the usefulness of a fair quantitative comparison, we have picked a target architecture component for which lots of optimizations have been proposed (data caches), and we have implemented most of the hardware data cache optimizations of the past 4 years in top conferences. Then we have ranked the different mechanisms, or more precisely, we have examined the impact of benchmark selection, process model precision,... on ranking, and obtained some surprising results. This study is part of a broader effort, called MicroLib, aimed at promoting the disclosure and sharing of simulator models.

1 Introduction

Simulators are used in most processor architecture research works, and, while most research papers include some performance measurements (often IPC and more specific metrics), these numbers tend to be distrusted because the simulator associated with the newly proposed mechanism is rarely publicly available, or at least not in a standard and reusable form, and as a result, it is not possible or easy to check for design and implementation hypotheses, potential simplifications or errors. However, since the goal of most processor architecture research works is to *improve* performance, i.e., do better than previous research works, it is rather frustrating not to be able to clearly quantify the benefit of a new architecture mechanism with respect to previously proposed mechanisms. Many researchers wonder, at some point, how their mechanism fares with respect to previously proposed ones and what is the best mechanism, at least for a given processor architecture and benchmark suite (or even a single benchmark); but many consider, with reason, that it is excessively time-consuming to implement a significant array of past mechanisms based on the articles only.

The purpose of this article is threefold: (1) to argue that, provided a few groups start populating a common library of modular simulator components, a broad and systematic quantitative comparison of architecture ideas may not be that unrealistic, at least for certain research topics and ideas; we introduce a library of modular simulator components aiming at that goal, (2) to illustrate this quantitative comparison using data cache research (and at the same time, we start populating the library), (3) to investigate the following set of methodology issues (in the context of data cache research) that researchers often wonder about but do not have the tools or resources to address:

- Which hardware mechanism is the best with respect to performance, power or cost?
- Are we making significant progress over the years?
- What is the impact of benchmark selection on ranking?
- What is the impact of the architecture model precision, especially the memory model in this case, on ranking?
- When programming a mechanism based on the article, does it often happen that we have to second-guess the authors' choices and what is the impact on mechanism performance and ranking?
- What is the impact of trace selection on ranking?

Comparing an idea with previously published ones means addressing two major issues: (1) how do we implement them? (2) how do we validate the implementations?

(1) The biggest obstacle to comparison is the necessity to implement again all the previously proposed and relevant mechanisms. Even if it usually means fewer than five mechanisms, we all know that implementing even a single mechanism can mean a few weeks of simulator development and debugging. And that is assuming we have all the necessary information for implementing it. Reverse-engineering all the implementation details of a mechanism from a 10-page research article can be challenging. An extended abstract is not really meant (or at least not usually written so as) to enable the reader to implement the hardware mechanism, it is meant to pass the idea, give the rationale and motivation, and convince the reader that it *can* be implemented; so some details are omitted because of paper space constraints or for fear they would bore the reader.

(2) Assuming we have implemented the idea presented in an article, then how do we validate the implementation, i.e., how do we know we have properly implemented it? First, we must be able to reconstruct exactly the same experimental framework as in the original articles. Thanks to widely used simulators like SimpleScalar [2], this has become easier, but only partially so. Many mechanisms require multiple minor control and data path modifications of the processor which are not always properly documented in the articles. Then, we need to have the same benchmarks, which is again facilitated by the Spec benchmarks [26], but they must be compiled with exactly the same compiler (e.g., the same *gcc* version) on the same platform. Third, we need to parameterize the base processor identically, and few of us specify all the SimpleScalar parameters in an article? Fortunately (from a reverse-engineering point of view) or unfortunately (from an architecture research point of view), many of us use many of the same default SimpleScalar parameters. Fourth, to validate an implementation, we need to compare the simulation results against the article numbers, which often means approximately reading numbers on a bar graph…And finally, since the first runs usually don't match, we have to do a combination of performance debugging and reverse-engineering of the mechanisms, based on second-guessing the authors' choices. By adding a dose of common sense, one can usually pull it off, but even then, there always remains some doubt, on apart of the reader of such a comparison, as to how accurately the researcher has implemented other mechanisms.

In this article, we illustrate these different points through data cache research. We have collected the research articles on performance improvement of data caches from the past four editions of the main conferences (ISCA, MICRO, ASPLOS, HPCA). We have implemented most of the mechanisms corresponding to pure hardware optimizations (we have not tried to reverse-engineer software optimizations). We have also implemented older but widely referenced mechanisms (*Victim Cache, Tag Prefetching* and *Stride Prefetching*). We have collected a total of 15 articles, and we have implemented only 10 mechanisms either because of some redundancies among articles (one article presenting an improved version of a previous one), implementation or scope issues. Examples of implementation issues are the *data compression prefetcher* technique [30] which uses data *values* (and not only addresses) which are not available in the base SimpleScalar version, *eager writeback* [16] which is designed for and tested on memory-bandwidth bound programs which were not available; an example of scope issue is the *non-vital loads* technique [20] which requires modifications of the register file, while we decided to focus our implementation and validation efforts on data caches only.

It is quite possible that our own implementation of these different mechanisms has some flaws, because we have used the same error-prone process described in previous paragraphs; so the results given in this article, especially the conclusion as to which are the best mechanisms, should be considered with caution. On the other hand, all our models are available on the MicroLib library web site [7], as well as the ranking, so authors or other researchers can check our implementation, and in case of inaccuracies or errors, we will be able to update the online ranking and the disseminated model.

Naturally, comparing several hardware mechanisms means more than just ranking them using various metrics. But the current situation is the opposite: researchers do analyze and compare ideas qualitatively, but they have no simple means for performing the quantitative comparisons.

This study is part of a broader effort called *MicroLib* which aims at facilitating the comparison and exchange of simulator models among processor architecture researchers. In Section 2 we present the *MicroLib* project, in Section 3 we describe our experimental framework, and in Section 4, we attempt to answer the questions listed above.

2 *MicroLib*

MicroLib. A major goal of MicroLib is to build an open library of processor simulator components which researchers can easily download either for directly plugging them in their own simulators, or at least for having full access to the source code, and thus to a detailed description

of the implementation. There already exists libraries of open simulator components, such as OpenCores [1], but these simulators are rather IP blocks for SoC (System-on-Chip), i.e., an IP block is usually a small processor or a dedicated circuit, while MicroLib aims at becoming a library of (complex) processor subcomponents (we will say processor *components* in the remainder of the article), and especially of various *research* propositions for these processor components.

Our goal is to ultimately provide researchers with a sufficiently large and appealing collection of simulator models that researchers actually start using them for performance comparisons, and more importantly, that they later on start contributing their own models to the library. As long as we have enough manpower, we want to maintain an up-to-date comparison (ranking) of hardware mechanisms, for various processor components, on the MicroLib web site. That would enable authors to demonstrate improvements to their mechanisms, to fix mistakes a posteriori, and especially, to provide the community with a clearer and fair comparison of hardware solutions for at least some specific processor components or research issues.

MicroLib and existing simulation environments. MicroLib modules can be either plugged into MicroLib processor models (a superscalar model called OoOSysC and a 15% accurate PowerPC750 model are already available [17]) which were developed in the initial stages of the project, or they can be plugged into existing processor simulators. Indeed, to facilitate the widespread use of MicroLib, we intend to develop a set of *wrappers* for interconnecting our modules with existing processor simulator models such as SimpleScalar, and recent environments such as Liberty [27]. We have already developed a SimpleScalar wrapper and all the experiments presented in this article actually correspond to MicroLib data cache hardware simulators plugged into SimpleScalar through a wrapper, rather than to our superscalar model. Next, we want to investigate a Liberty wrapper because some of the goals of Liberty fit well with the goals of MicroLib, especially the modularity of simulators and the planned development of a library of simulator modules. Rather than competing with modular simulation environment frameworks like Liberty (which aim at providing a full environment, and not only a library), we want MicroLib to be viewed as an open and, possibly federating, project that will try to build the largest possible library through extensive wrapper development. There are also many modular environments in the industry, such as ASIM [5] by Compaq (and now Intel), and though they are not publicly available, they may benefit from the library, provided a wrapper can be developed for them. The current MicroLib modules are based on *SystemC* [19], a modular simulation framework supported by more than 50 companies from the embedded domain, which is quickly becoming a *de facto* standard in the embedded world for cycle-level or more abstract simulation. All the mechanisms presented in this article were implemented using *SystemC*.

MicroLib modules and design guidelines for SystemC. SystemC bears many similarities with Liberty again as it provides a software support for building modules, links between modules and an event engine. On the other hand, it is a bare environment as it specifies no guideline for implementing modules and communication protocols between modules. The reason for such freedom is the very large range of applications of SystemC. This environment can be used either for Transaction-Level Modeling (TLM), where only the module functions are described with very rough performance estimates, for cycle-level simulation, and VHDL/Verilog modules can even be wrapped within SystemC modules and combined with other more abstract components models. To implement these possibilities, SystemC offers a rather large range of communication methods: the most simple is the *Signal* which is similar to a physical link (either a bit or a set of bits), and there are also *Channels* for more elaborate link behavior, and even *Events* where physical links disappear. Because we target cycle-level simulation, we only use *Signals* for communications among modules.

Modular simulator design may not significantly speed up the development of a new simulator, but it considerably speeds up the modifications and updates of an existing simulator (and that is the most frequent task in a research group), because most modifications are local to one or a few modules, and a clean representation of communications among modules (through links) provides an instant and intuitive representation of the relationships among modules (processor components). However modular simulators are significantly slower than monolithic simulators, typically a factor of 10 to 15; for instance, our OoOSysC superscalar model executes 25000 instructions per cycle on an Athlon XP 1800+, while SimpleScalar executes 300000 instructions per cycle. However our experience is that we spend much more time in simulator development than in simulation runs within a research project. And recent sampling techniques like SimPoint [22] and SMARTS [28] have shown that it is possible to reduce simulation time by several orders of magnitude.

In fact, striking the right balance between modularity, efficiency and speed is a delicate task. A too fine-grain granularity and the simulator is close to the architecture, but the code is excessively large and slow; a too coarse-

grain granularity and the benefits of modular simulation are lost. Our initial OoOSysC implementation had 29 modules, and we have progressively decreased it to 25 modules (at this level, one pipeline stage roughly corresponds to one or a few modules), both for software engineering and performance reasons.

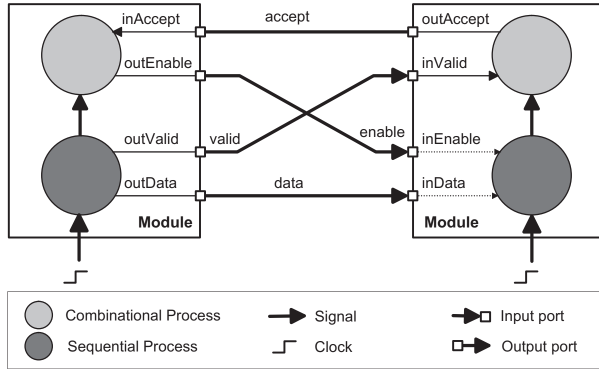

Figure 1: *Modular structures of MicroLib.*

The performance price is due to two factors: the communication overhead and processes wake-ups. The communication overhead comes from the fact that exchanging an information between two hardware components in a monolithic simulator just means reading a variable, while in a module simulator it means writing to an output port, waking up a link, writing to an input port, waking up a module, reading the input port. The number of times a module is waken up is the second performance factor. Consider a 2-input module for instance, and assume the module receives the two inputs from two different sources within the same cycle; then, the module will be waken up upon arrival of each input, but it is only after the second wake-up that it can produce the result; in fact the first wake-up is useless. For that purpose, we have defined communication protocols, on top of SystemC, that minimize the number of wake-ups in order to ensure reasonable performance. In Liberty for instance, the communication protocols are embedded in the environment, while in SystemC, they have to be explicited; but the development overhead is fairly small. Figure 1 shows the relationships and links between two modules. The main guideline is to split modules into two parts: one that will be waken up every clock cycle (called sequential processes), and one that will be waken up if incoming signals change (called combinational processes). Combinational processes can be the costliest because they can be waken up several times per cycle, so they are limited in numbers and their actions as far as possible.

3 Experimental Framework

Parameter	Value
Processor core	
Processor Frequency	2 GHz
Instruction Windows	128-RUU, 128-LSQ
Fetch, Decode, Issue width	8 instructions per cycle
Functional units	8 IntALU, 3 IntMult/Div, 6 FPALU, 2 FPMult/Div, 4 Load/Store Units
Commit width	up to 8 instructions per cycle
Memory Hierarchy	
L1 Data Cache	32 KB/direct-mapped
L1 Data Write Policy	Writeback
L1 Data Allocation Policy	Allocate on Write
L1 Data Line Size	32 Bytes
L1 Data Ports	4
L1 Data MSHRs	8
L1 Data Reads per MSHR	4
L1 Data Latency	1 cycle
L1 Instruction Cache	32 KB/4-way associative/LRU
L1 Instruction Latency	1 cycle
L2 Unified Cache	1 MB/4-way associative/LRU
L2 Cache Write Policy	Writeback
L2 Cache Allocation Policy	Allocate on Write
L2 Line Size	64 Bytes
L2 Ports	1
L2 MSHRs	8
L2 Reads per MSHR	4
L2 Latency	12 cycles
L1/L2 Bus	32-byte wide, 2 Ghz
Bus	
Bus Frequency	400 MHz
Bus Width	64 bytes (512 bits)
SDRAM	
Capacity	2 GB
Banks	4
Rows	8192
Columns	1024
RAS To RAS Delay	10 cpu cycles
RAS Active Time	80 cpu cycles
RAS to CAS Delay	15 cpu cycles
CAS Latency	10 cpu cycles
RAS Precharge Time	15 cpu cycles
RAS Cycle Time	55 cpu cycles
Refresh	Avoided
Controler Queue	32 Entries

Table 1: *Baseline configuration.*

3.1 SystemC and SimpleScalar

As mentioned before, for all the experiments of this article, our MicroLib data cache modules are plugged into SimpleScalar. Two reasons motivated this choice. First, all the mechanisms, except for *Frequent Value Cache* [31], *Markov Prefetching* [12] and *Content-Directed Data Prefetching* [3] , were implemented using SimpleScalar, and it is easier to validate the implementation if we use the same processor simulator. Second, we wanted to show that MicroLib modules developed in SystemC can be plugged into existing simulators through a wrapper (exactly an interface in this case). For that purpose, we have stripped SimpleScalar of its cache and memory models, and replaced them with MicroLib models. In addition to the various data cache models, we have developed and used an SDRAM model for most experiments. Note that more detailed memory models have been recently made available for SimpleScalar [2].

We have used SimpleScalar 3.0d [2] and the parameters in Table 1 which we found in many of the target articles [15, 10, 9]; they correspond to a scaled up superscalar implementation (note the bus width is rather large, for instance); the other parameters are set to their default values.

We have compared the mechanisms using the SPEC CPU2000 benchmark suite [26]. The benchmarks were compiled for the Alpha instruction set using `cc` DEC C V5.9-008 on Digital UNIX V4.0 (Rev. 1229), `cxx` Compaq C++ V6.2-024 for Digital UNIX V4.0F (Rev. 1229), `f90` Compaq Fortran V5.3-915 and `f77` Compaq Fortran V5.3-915 compilers with SPEC peak settings. For each program, we fastforwarded 1 billion instructions, and then simulated 2 billion instructions with the reference input set.

3.2 Validating the Implementation

Validating a hybrid SimpleScalar+MicroLib model. Because we plugged our own cache simulator into SimpleScalar, we wanted to validate the hybrid SimpleScalar+MicroLib model against the original SimpleScalar model, in order to show that the hybridation introduces minimal noise. Our cache architecture choices are different, and we believe more realistic, than in SimpleScalar. For the validation, we have altered the SimpleScalar model so that it ressembles ours and validated this altered SimpleScalar model against the SimpleScalar+MicroLib model; in order to validate specifically the cache, we have used the SimpleScalar memory model in both simulators. In Section 4.3, we analyze the impact of the memory model accuracy.

Initially, before altering the SimpleScalar cache model, we found a 6.8% IPC difference in average between the hybrid implementation and the original SimpleScalar implementation. We then progressively modified the SimpleScalar cache model to get closer to our MicroLib model and found that most of the performance variation is due to the following implementation differences:

- The SimpleScalar MSHR (miss address file [14, 24]) has unlimited capacity; in our cache model its capacity parameters are defined in Table 1.
- In SimpleScalar, the cache pipeline is insufficiently detailed. As a result, a cache request can never delay next requests, while in a pipelined implementation, such delays can occur. Several events can delay a request: two misses on the same cache line but for different addresses can stall the cache, upon receiving a request the MSHR is not available for one cycle...
- The processor Load/Store Queue (LSQ) can always send requests to the cache in SimpleScalar, while the abovementioned cache stalls (plus MSHR full) can temporarily stall the LSQ.
- In SimpleScalar, a dirty line is evicted while in the same cycle, the miss request is sent to the lower level; the litterature suggests both actions usually take place in separate cycles [8].
- In SimpleScalar the refill requests (incoming memory request) seem to use additional cache ports. For instance, when the cache has two ports, it is possible to have two fetch requests and a refill request at the same time. We strictly enforce the number of ports, and upon a refill request, only one normal cache request can occur with two ports.

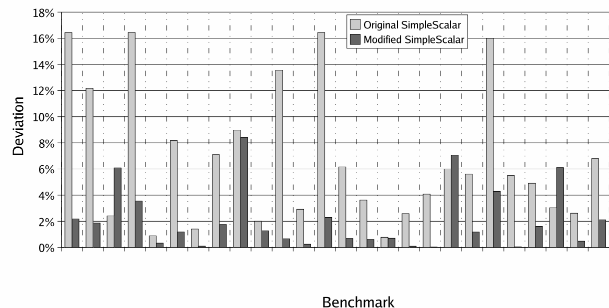

Figure 2: *MicroLib cache model validation.*

After altering the SimpleScalar model so it behaves like our MicroLib model, we found that the average IPC difference between the two models was down to 2%, see Figure 2. Note that, in the remainder of the article, we *do not* use the SimpleScalar model, we use our original and unmodified MicroLib model.

Besides this performance validation, we have done additional correction validations using the OoOSysC superscalar processor. We plugged our different models in OoOSysC which has the additional advantage of actually performing all computations. As a result, the cache not only contains the addresses but the *actual values* of the data, i.e., it really executes the program, unlike SimpleScalar. Comparing the value in the emulator and the simulator for every memory request is a simple but powerful debugging tool.[1] For instance, in one of the implemented models, we forgot to properly set the dirty bit in some cases; as a result, the corresponding line was not systematically written back to memory, and at the next request at that address, the values differed.

Validating the implementation of data cache mechanisms. The most time-consuming part of this research work was naturally reverse-engineering the different hardware mechanisms from the research articles. The different mechanisms, a short description and the corresponding reference are listed in Table 2, and the mechanism-specific parameters are listed in Table 3.

For several mechanisms, there was no easy way to do an IPC validation. The metric used in *FVC* and *Markov* is miss ratio, so only a miss ratio-based validation was possible. *VC*, *Tag* and *SP* have been proposed several years ago, so the benchmarks and the processor model differed significantly. *CDP* and *CDPSP* used an internal Intel simulator and their own benchmarks. For all the above mechanisms, the validation consisted in ensuring that absolute performance values were in the same range, and that tendencies were often similar (relative performance difference of architecture parameters, among benchmarks, etc. . .).

For *TK*, *TKVC*, *TCP* and *DBCP*, we used the IPC graphs provided in the articles for the validation; the benchmarks used in each article are indicated in Table 4. Figure 3 shows the percentage speedup difference between the graph numbers and our simulations (some articles do not provide IPC, but only speedups with respect to the base SimpleScalar cache configuration). The average error is 5%, but the difference can be very significant for certain benchmarks, especially *ammp*. We were not able to bridge this performance difference even though we tested many values of the unspecified (undocumented) parameters. In general, tendencies are preserved, but not always, i.e., a speedup or a slowdown in an article can become a slowdown or a speedup in our experiments, as for *gcc* (for *TK* and *DBCP*) and *gzip* (for *TK*) respec-

[1] Besides debugging purposes, this feature is also particularly useful for testing value prediction mechanisms.

Parameter	Value
Victim Cache	
Size/Associativity	512 Bytes / Fully assoc.
Frequent Value Cache	
Number of lines	1024 lines
Number of frequent values	7 + unknow value
Timekeeping Cache	
Size/Associativity	512 Bytes/Fully assoc.
TK refresh	512 cpu cycles
TK threshold	1023 cycles
Markov Prefetcher	
Prediction Table Size	1 MB
Predictions per entry	4 predictions
Request Queue Size	16 entries
Prefetch Buffer Size	128 lines (1 KB)
Tag Prefetching	
Request Queue Size	16
Stride Prefetching	
PC entries	512
Request Queue Size	1
Content-Directed Data Prefetching	
Prefetch Depth Threshold	3
Request Queue Size	128
CDP + SP	
SP PC entries	512
CDP Prefetch Depth Threshold	3
Request Queue Size (SP/CDP)	1/128
Timekeeping Prefetcher	
Address Correlation	8KB, 8-way assoc.
Request Queue Size	128 entries
Tag Correlating Prefetching	
THT size	1024 sets, direct-mapped, stores 2 previous tags
PHT size	8KB, 256 set, 8 way assoc.
Request Queue Size	128 entries
Dead-Block Correlating Prefetcher	
DBCP history	1K entries
DBCP size	2M 8-way
Request Queue Size	128 entries
Global History Buffer	
IT entries	256
GHB entries	256
Request Queue Size	4

Table 3: *Configuration of data cache optimizations.*

Acronym	Mechanism	Description
VC	Victim Cache [13]	A small fully associative cache associated for storing evicted lines; particularly useful for limiting the impact of conflict misses without resorting to associativity.
FVC	Frequent Value Cache [31]	A small additional cache that behaves like a victim cache, except that it is just used for storing frequently used values in a compressed form. The technique has also been applied in other studies [30, 29] to prefetching and energy reduction.
TK	Timekeeping [9]	Prefetch mechanism that time statistics to estimate when a cache line is about to be replaced and prefetches the new address for that line.
TKVC	Timekeeping Victim Cache [9]	Same as TK but uses a victim cache instead of prefetching.
Markov	Markov Prefetcher [12]	Uses Markov chains to determine prefetch addresses.
TP	Tag Prefetching [25]	A very simple prefetching technique that prefetches on a miss, or on a hit on a prefetched line.
SP	Stride Prefetching [13]	An extension of tag prefetching that detects the access stride of load instructions and prefetches accordingly.
CDP	Content-Directed Data Prefetching [3]	A prefetch mechanism for pointer-based data structures that attempts to determine if a fetched line is actually an address, and if so, prefetches it immediately.
CDPSP	CDP + SP	A combination of CDP and SP as proposed in [3].
TCP	Tag Correlating Prefetching [10]	Prefetcher that correlates cache misses to generate prefetches.
DBCP	Dead-Block Correlating Prefetcher [15]	A prefetcher that, like TK, predicts when a line will be replaced and by which address. It detects a line that is about to be evicted by the addresses of load/store instructions accessing it.
GHB	Global History Buffer [18]	We implemented only one of the possible variations which determines a stride for prefetching, like SP, except that the stride is computed based on a history of misses.

Table 2: *Target data cache optimizations.*

Mechanism	ammp	applu	apsi	art	equake	facerec	fma3d	galgel	lucas	mesa	mgrid	sixtrack	swim	wupwise	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	twolf	vortex	vpr
DBCP	√			√	√														√		√				
TK/TKVC/TCP/DBCPTK	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
GHB	√			√				√	√		√		√	√	√			√			√	√			√

Table 4: *Benchmarks used in validated mechanisms.*

tively. Note that, surprisingly enough, all four mechanisms use exactly the same SimpleScalar parameters of Table 1, even though the first mechanism was pulished in 2000 and the last one in 2003. Only the SimpleScalar parameters of *GHB* (not included in the graph of Figure 3), proposed at HPCA 2004, are different (130 cycles memory latency).

Finally, note that the accuracy of *DBCP* is rather poor, while it is much higher for *DBCPTK*; the *DBCPTK* values have been extracted from the article which proposed *TK* [9] and which compared *TK* against *DBCP*. Interestingly, their own reverse-engineering effort brought almost the same results as ours, but both are fairly different from the original article, outlining the difficulty of an accurate reverse-engineering process.

4 A Quantitative Comparison of Hardware Data Cache Optimizations

The different subsections correspond to the questions listed in Section 1. Except for Section 4.1, all the comparisons relate to the IPC metric and are usually averaged over all the benchmarks listed in Section 3.1, except for Section 4.2.

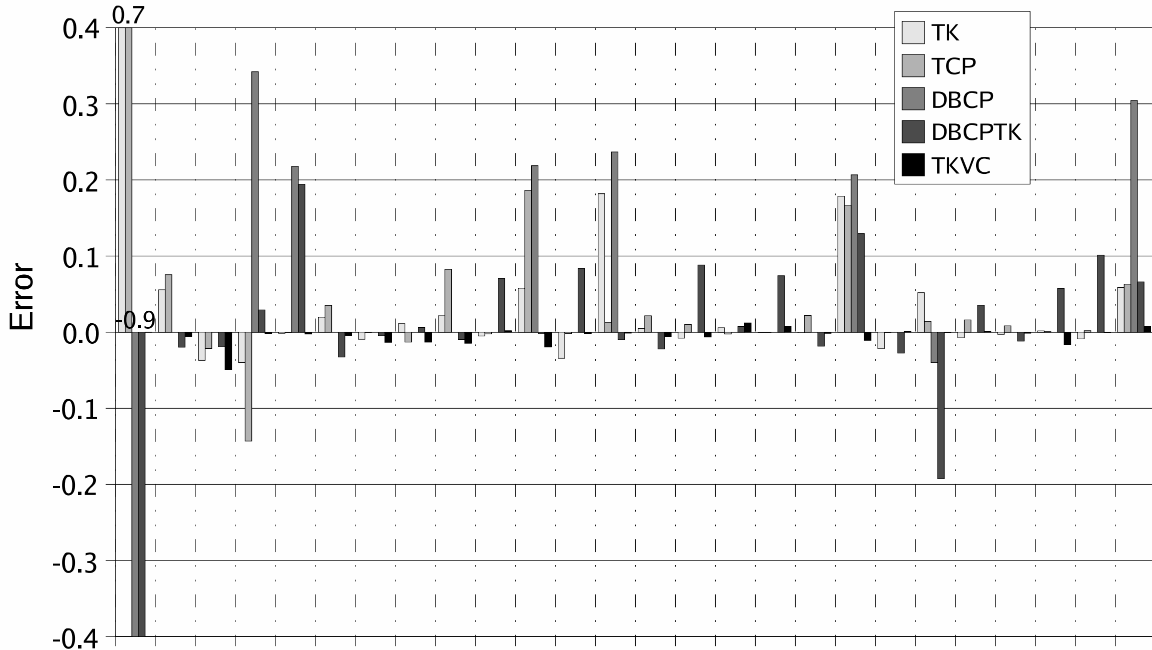

Figure 3: *Validation of TK, TCP, DBCP and TKVC.*

4.1 Which hardware mechanism is the best with respect to performance, power and/or cost? Are we making any progress?

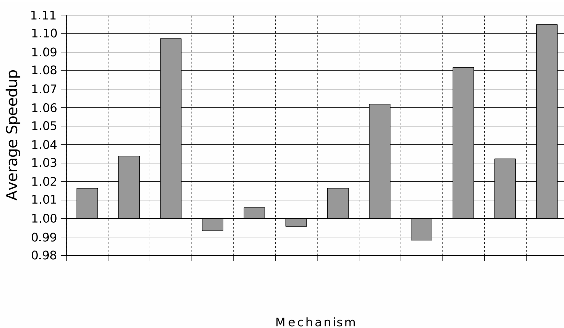

Figure 4: *Speedup.*

Performance. Figure 4 shows the average IPC speedup over the 26 benchmarks for the different mechanisms with respect to the base cache parameters defined in Section 3.1.[2] We find that the best mechanism is *GHB*, a recent evolution (HPCA 2004) of *SP*, an idea originally published in 1990, and which is the second best performing mechanism, then followed by *TK*, proposed in 2002. A very simple (and old) hardware mechanism like *TP* performs also quite well. Comparably, more recent ideas like *TCP* or *DBCP* exhibit rather disappointing performance, and *FVC*, which was evaluated using miss ratios in the article, seems to provide little IPC improvements. Overall, it is striking to observe how irregularly performance has evolved from 1990 to 2004, when all mechanisms are considered within the same processor.

Note that the speedup for some of the mechanisms in Figure 4 is fairly close to the reverse-engineering error shown in Figure 3, meaning that the validity of the comparison itself may be jeopardized by the necessity to reverse-engineer mechanisms.

Cost. We evaluated the relative cost (chips area) of each mechanisms using CACTI 3.2 [23], and Figure 5

[2]The IPC graphs per benchmark are available online at `http://www.microlib.org`.

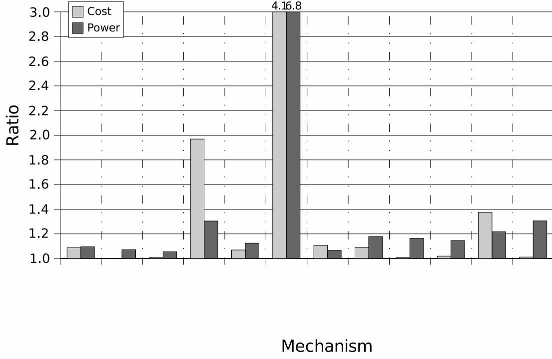

Figure 5: *Power and Cost Ratios.*

provides the area ratio (relative cost of mechanism with respect to base cache). Not suprisingly, *Markov* and *DBCP* have very high cost due to large tables, while other lightweight mechanisms like *TP*, or even *SP* and *GHB* (small tables) incur almost no additional cost. What is more interesting is the correlation between performance and cost: *GHB* and *SP* remain clear winners, and *TP* is more attractive in that perspective. On the other hand, *DBCP*, which performs slightly better than *TP*, does not compare favorably.

Power. We evaluated power using XCACTI [11]; Figure 5 shows the relative power increase of each mechanism. Naturally, power is determined by cache area and activity, and not surprisingly, *Markov* and *DBCP* have strong power requirements. In theory, a costly mechanism can compensate the additional cache power consumption with more efficient, and thus reduced cache activity, though we found no clear example along that line. Conversely a cheap mechanism with significant activity overhead can be power greedy. It is apparently the case for *GHB*: even though the additional table is small, each miss can induce up to 4 requests, and a table is scanned repeatedly, hence the high power consumption. In *SP*, on the other hand, each miss request induces a single request, and thus *SP* is very efficient, just like *TP*.

Best overall tradeoff (performance, cost, power). When power and cost are factored in, *SP* seems like a clear winner, *TK* and *TP* performing also very well. *TP* is the oldest mechanism, *SP* has been proposed in 1990 and *TK* has been very recently proposed in 2002. While which mechanism is the best very much depends on industrial applications (e.g., cost and power in embedded processors, versus performance and power in general-purpose processors), it is probably fair to say that the progress of data cache research over the past 15 years has been all but regular.

In the remaining sections, ranking is focused on performance due to paper space constraints, but naturally, it would be necessary to come up with similar conclusions for power, cost, or all three parameters combined.

DBCP vs. Markov
TKVC vs. VC
TK vs. DBCP
CDP/CDPSP vs. SP
TCP vs. DBCP
GHB vs. SP

Table 5: *Previous comparisons.*

Did the authors compare their ideas? Table 5 shows which mechanism has been compared to which previous mechanisms (listed in chronological order). Most of the articles have few if no quantitative comparison with previous mechanisms, except when comparisons are almost compulsory, like *GHB* which compares against *SP* because it is based on *SP*. Sometimes, comparisons are performed against the most recent mechanism, maybe with the expectation it is the current best one, like *TCP* and *TK* which are compared against *DBCP*, while in this case, a comparison with *SP* might have been more appropriate.

4.2 What is the impact of benchmark selection on ranking?

Yes, cherry-picking is wrong. We have ranked the different mechanisms for every possible benchmark combination. First, we have observed that for any number of benchmarks less or equal than 23, i.e., the average IPC is computed over 23 benchmarks or less, there is always *more than one winner*, i.e., it is always possible to find two benchmark selections with different winners. In Figure 6, we have indicated how often a mechanism can be a winner for any number of benchmarks up to 26. For instance, mechanisms that perform poorly on average, like *CDP*, can win for selections of up to 2 benchmarks; note that *CDP* is a prefetcher for pointer-based data structures, so that it is likely to perform well for benchmarks with many misses in such data structures; for the same reason, *CDPSP* (a combination of *SP* and *CDP*) can be appropriate for a larger range of benchmarks, as the authors point out. Another astonishing result is *Markov* which can perform very well for up to 6-benchmark selections.

Are there "representative" benchmarks? We could not find a single benchmark for which the ranking is the same as when computed over the full 26 benchmarks. The

	Base	VC	TP	SP	Markov	FVC	DBCP	TKVC	TK	CDP	CDPSP	TCP	GHB
1		√	√	√	√	√	√	√	√	√	√		√
2		√	√	√	√	√	√	√	√	√	√		√
3		√	√	√	√	√	√	√	√		√		√
4		√	√	√	√	√	√	√	√		√		√
5		√	√	√	√	√	√	√	√		√		√
6		√	√	√	√	√		√	√		√		√
7		√	√	√	√	√		√	√		√		√
8		√	√	√	√	√		√	√		√		√
9		√	√	√	√	√		√	√		√		√
10		√	√	√		√		√	√		√		√
11		√	√	√		√		√	√		√		√
12		√	√	√		√		√	√		√		√
13		√	√	√				√	√		√		√
14		√	√	√				√	√		√		√
15		√	√	√				√	√		√		√
16		√	√	√				√	√		√		√
17			√	√				√	√		√		√
18			√	√				√	√		√		√
19			√	√				√	√		√		√
20				√					√		√		√
21				√					√		√		√
22				√					√		√		√
23				√									√
24													√
25													√
26													√

Table 6: *Which mechanism can be winner with x benchmarks?*

size of the smallest "representative" benchmark selection we found is 6. There are several such 6-benchmark representative selections; an example is the set *ammp, applu, apsi, art, mesa, crafty*.

4.3 What is the impact of the architecture model precision on ranking?

Figure 6: *Impact of the memory model accuracy.*

Is it necessary to have a more detailed memory model? We have implemented a detailed SDRAM model, as Cuppu et al. [4] did for SimpleScalar (though their model is not yet distributed), and we have evaluated the influence of the memory model on ranking. The original SimpleScalar memory model is rather raw with a constant memory latency. Our model uses a bank interleaving scheme [21, 32] which allows the DRAM controller to hide the access latency by pipelining page opening and closing operations. We implemented several schedule schemes proposed by Green et al. [6] and retained one that significantly reduces conflicts in row buffers. For the sake of the comparison with the 70-cycle SimpleScalar memory, we have scaled down the parameters of our PC133 SDRAM, see Figure 1, to reach an *average* 70 cycles over all benchmarks. Figure 6 compares this memory model with a SimpleScalar-like memory model. The memory model does affect significantly, if not considerably, the absolute performance as well as the ranking of the different mechanisms. The most dramatic reduction occurs for *GHB* which drops from a 1.19 speedup with a SimpleScalar-like memory to less than 1.11 with an SDRAM memory; the performance advantage of *GHB* over *SP* is considerably smaller with a more realistic memory because *GHB* increases considerably the memory preassure. The memory model also affects ranking: for instance, *CDPSP* outperforms *SP* with a simplified memory model and no longer with an SDRAM; the same is true of *VC* and *DBCP*...

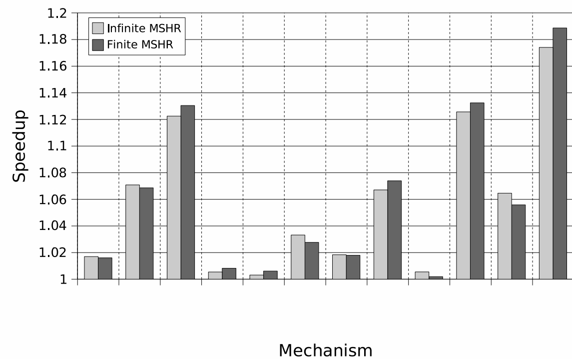

Figure 7: *Impact of the cache model accuracy.*

Influence of cache model inaccuracies. Similarly, we have investigated the influence of other hierarchy model components. For instance, we have explained in Section 3.2, that the SimpleScalar cache uses an infinite miss address file (MSHR), so we have compared the impact of just varying the miss address file (i.e., infinite versus the baseline value defined in Table 1). Figure 7 shows that for many mechanisms, the MSHR has limited impact

on performance and ranking, except for *CDP*, because it strongly increases MSHR blocking situations in this case; with an infinite MSHR, *CDP* is the eleventh mechanism, close to *Markov*, then drops to the last rank with a finite MSHR.

4.4 What is the impact of second-guessing the authors' choices?

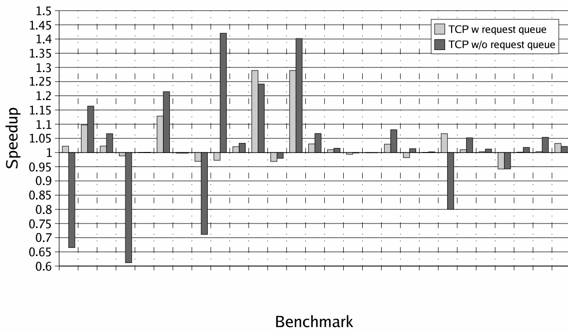

Figure 8: *Impact of second-guessing the authors' choices.*

For several of the mechanisms, some of the implementation details were missing in the article, or the interaction between the mechanisms and other components were not sufficiently described, so we had to second-guess them. While we cannot list all such omissions, we want to illustrate their potential impact on performance and ranking, and that they can significantly complicate the task of reverse-engineering a mechanism.

One such case is *TCP*; the article properly describes the mechanism, how addresses are predicted, but it gives few details on how and when prefetch requests are sent to memory. Among the many different possibilities, prefetch requests can be buffered in a queue until the bus is idle and a request can be sent. Assuming this buffer effectively exists, a new parameter is the buffer size; it can be either 1 or a large number (we ended up using a 128-entry buffer), and the buffer size is a tradeoff, since a too short buffer size will result in the loss of many prefetch requests, and a too large one may excessively delay some prefetch requests. Figure 8 shows the performance difference and ranking for a 128-entry and a 1-entry buffer. All possible cases are found: for some benchmarks like *mgrid* and *swim*, the performance difference is tiny, while it is dramatic for *art, lucas* and *galgel*.

We ended up selecting 128 because it matched best the average performance presented in the article, though it is quite possible the authors did not actually use such a buffer (and used another unguessed variation). This is just one example among the many difficulties which were part of the reverse-engineering process.

4.5 What is the impact of trace selection on ranking?

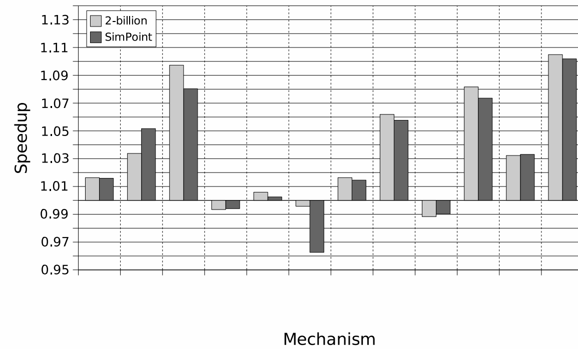

Figure 9: *Impact of trace selection.*

Most researchers tend to skip an arbitrary (usually large) number of instructions in a trace, then simulate the largest possible program chunk (usually of the order of a few hundred million to a few billion instructions), as we have done ourselves in the present article. Sampling has received increased attention in the past few years, with the prospect of finding a robust and practical technique for speeding up simulation while ensuring the representativity of the sampled trace. The most notable and practical contribution is SimPoint [22] which showed that a small trace can highly accurately describe a whole program behavior.

We used the SimPoint tools to generate the basic block vectors (BBV) for a 500-million trace for each program. Then, we compared the impact of trace size selection: our "skip 1 billion, simulate 2 billion" trace versus SimPoint trace. Figure 9 shows the average performance achieved with each method, and they differ significantly. For instance *DBCP* performance decreases significantly and it is now the worse mechanism instead of *CDP*, and overall most mechanisms perform worse, with the notable exception of *TP*. Not surprisingly, trace selection can have a considerable impact on *research* decisions like selecting the most appropriate mechanism, and obviously, even large 2-billion traces do not constitute a sufficient precaution.

5 Conclusions and Future Work

In this article we have illustrated with data caches the MicroLib approach for enabling the quantitative comparison of hardware optimizations. We have implemented several recent hardware data cache optimizations and we have shown that many methodology variations or flaws can result in an incorrect assessment of what is the best or most appropriate mechanism for a given architecture. Our goal is now to populate the library, to encourage the quantitative comparison of mechanisms, and to maintain a regularly updated comparison (ranking) for various hardware components.

References

[1] OPENCORES. http://www.opencores.org, 2001-2004.

[2] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, Department of Computer Sciences, University of Wisconsin, June 1997.

[3] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 279–290, San Jose, California, October 2002.

[4] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A performance comparison of contemporary dram architectures. In *Proceedings of the 26th annual international symposium on Computer architecture (ISCA)*, pages 222–233, Atlanta, Georgia, United States, June 1999.

[5] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubbendu S. Mukkerjee, Harish Patil, Steven Wallace, Nathan Binkert, and Toni Juan. ASIM: A performance model framework. In *IEEE Computer, Vol. 35, No. 2*, February 2002.

[6] Christian Green. Analyzing and implementing SDRAM and SGRAM controllers. In *EDN (www.edn.com)*, February 1998.

[7] Alchemy Research Group. MicroLib. http://www.microlib.org, 2001-2004.

[8] Jim Handy. *The Cache Memory Book*. Academic Press, 1993. HAN j 98:1 1.Ex.

[9] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings of the 29th annual international symposium on Computer architecture (ISCA)*, pages 209–220, Anchorage, Alaska, May 2002.

[10] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. TCP: Tag correlating prefetchers. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA)*, Anaheim, California, February 2003.

[11] M. Huang, J. Renau, S. M. Yoo, and J. Torrellas. L1 data cache decomposition for energy efficiency. In *International Symposium on Low Power Electronics and Design (ISLPED 01)*, Huntington Beach, California, August 2001.

[12] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture (ISCA)*, pages 252–263, Denver, Colorado, United States, June 1997.

[13] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. Technical report, Digital, Western Research Laboratory, Palo Alto, March 1990.

[14] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 18th International Symposium on Computer Architecture*, Toronto, Canada, May 1981.

[15] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th annual international symposium on Computer architecture (ISCA)*, pages 144–154, Gteborg, Sweden, June 2001.

[16] Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 11–21. ACM Press, 2000.

[17] G. Mouchard. PowerPC G3 simulator. http://www.microlib.org/G3/PowerPC750.php, 2002.

[18] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, page 96, Madrid, Spain, February 2004.

[19] OSCI. SystemC. http://www.systemc.org, 2000-2004.

[20] Ryan Rakvic, Bryan Black, Deepak Limaye, and John P. Shen. Non-vital loads. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*. ACM Press, 2002.

[21] Tomas Rockicki. Indexing memory banks to maximize page mode hit percentage and minimize memory latency. Technical report, HP Laboratories Palo Alto, June 1996.

[22] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Tenth international conference on architectural support for programming languages and operating systems on Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 45–57. ACM Press, 2002.

[23] Premkishore Shivakumar and Norman P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical report, HP Laboratories Palo Alto, August 2001.

[24] James Edwards Sicolo. *A Multiported Nonblocking Cache For a Superscalar Uniprocessor*. Phd. thesis, B.S., State University of New York, Buffalo, 1989.

[25] Alan J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[26] SPEC. SPEC2000. http://www.spec.org.

[27] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, Istanbul, Turkey, November 2002.

[28] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97. ACM Press, 2003.

[29] Jun Yang and Rajiv Gupta. Energy efficient frequent value data cache design. In *Proceedings of the 35th international symposium on Microarchitecture (MICRO)*, pages 197–207, Istanbul, Turkey, November 2002.

[30] Youtao Zhang and Rajiv Gupta. Enabling partial cache line prefetching through data compression. In *International Conference on Parallel Processing (ICPP)*, Kaohsiung, Taiwan, October 2003.

[31] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems (ASPLOS-IX)*, pages 150–159, Cambridge, Massachusetts, United States, November 2000.

[32] Zhao Zhang, Zhlichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd international symposium on Microarchitecture (MICRO)*, Monterey, California, December 2000.