On the Impact of OS and Linker Effects on Level-2 Cache Performance

Hans Vandierendonck Koen De Bosschere
Ghent University
Dept. of Electronics and Information Systems
St.-Pietersnieuwstraat 41
B-9000 Gent, Belgium
{hvdieren,kdb}@elis.ugent.be

Abstract

The design of microprocessors depends strongly on architectural simulation. As simulation can be very slow, it is necessary to reduce simulation time by simplifying the simulator and increasing its level of abstraction. A very common abstraction is to ignore operating system effects. As a result of this, there is no information available during simulation about the relationship between virtual addresses and physical addresses This information is important for lower-level caches and main memory as these memories are indexed using the physical address. Another simplification relates to simulating only statically linked programs, instead of the commonly used dynamic linking. This results in different data layouts and, as we show in this paper, it effects the miss rate of physically indexed caches such as the level-2 cache. This paper investigates the error associated to these simplifications in the modeling of level-2 caches and shows that performance can be underestimated or overestimated with errors up to 24%.

1 Introduction

The performance of microprocessors is evaluated using simulation during all phases of the design process. Because simulation is slow and implementing detailed models is complex and time-consuming, it is very common that the accuracy of a simulator is low. Furthermore, simulators are rarely validated against real machines. This situation is dangerous, as deciding whether a new feature is useful for a microprocessor using inaccurate simulators can easily lead to wrong results: good features can be missed and useless features can appear good [3, 8]. Accurate simulators are thus extremely important for both industry and academia.

One simplification to a simulation infrastructure is made very frequently due to the high cost of implementing it: ignoring the operating system. This abstraction is present in SimpleScalar [2], RSIM [18] SMTSIM [23], and MINT [26]. Instead of simulating the operating system, system calls are trapped by the simulator and are handed over to the host operating system. This modeling is generally expected to be accurate when simulating programs that do not spend a large amount of time in the operating system. However, several studies have shown that this assumption is false, even when considering single-thread performance. E.g. it is extremely important to accurately model the time it takes for the operating system to handle a TLB miss when a program thrashes the TLB [8]. It is also important to model cache control instructions executed by the operating system, as these may avoid cache misses [4]. Small pieces of OS code may displace data from caches [22] or modify branch prediction information [9, 14].

In this paper, we analyze the effect on performance of two other properties of the operating system: the translation of virtual to physical addresses and the use of static versus dynamic linking of programs. These effects interact with the layout of code and data and determine the performance of the memory system, such as conflict misses in the level-2 cache and bank conflicts in the memory system.

The translation of virtual to physical addresses is performed by the operating system as part of implementing the virtual memory system [11]. A virtual address points to a programs private address space. A physical address points into the physical address space, shared by all programs and managed by the operating system. The virtual address is translated to a physical address for every memory access made. The mapping is determined by the operating system but cached in a hardware structure called translation lookaside buffer (TLB) to allow fast translation.

The virtual address translation is foremost constrained by the fact that the operating system must try to hold each program's working set in main memory. In principle, page replacement policies are similar to replacement policies for caches: older pages are swapped out when a new page is brought in and new pages can be placed only in empty

frames. The OS can in principle map a virtual address to almost any physical address it desires thereby changing the distribution of accesses to the sets of a physically indexed cache. Thus, the access pattern of virtual addresses may be skewed more or less than the access pattern composed of the corresponding physical addresses.

Simulators that ignore the operating system have no information available on the physical addresses and assume that the physical addresses equal the virtual addresses. This abstraction introduces an unknown error into the simulation as one can observe a different number of cache misses depending on the type of addresses used to access the cache.

Simulating statically linked programs (i.e. all libraries are embedded in the program binary) because this simplifies the loader and increases the ease-of-use of the simulation environment. However, high-performance systems almost exclusively use dynamic linking: the libraries are linked in only when the program is started. There are many reasons for this, e.g. memory resources are saved as dynamic library code can be shared among multiple programs, improved versions of libraries are available without having to relink, etc. The unrealistic choice of simulating statically linked programs again introduces an unknown error.

Besides introducing errors in the simulation, the effects above also introduce variability in the performance metric. E.g. the virtual address translation is different during each run of a program, causing a different number of conflict misses to occur. Indeed, the mapping depends on the unused page frames in the system at the time the program demands new pages. The unused page frames depend on the system activity before the program starts execution, which is different each time the program is run. The virtual-to-physical mapping may also change during the execution of a program. The rate at which it changes depends on the number of pages that are requested by this application, as well as other activity in the system. We measure this variability and take it into account when evaluating the performance of the memory hierarchy.

Besides showing that virtual address translation and linking mode affect absolute performance, we also show that these effects can **misjudge the utility of an optimization** to the processor. Instead of predicting a speedup, the real system may not speed up or worse, it may even slow down. As a case study, we investigate how these two parameters interact with the addition of a hash function in the level-2 cache. A hash function tries to randomize the stream of cache accesses evenly among all sets of the cache such that conflict misses are minimized [10, 13, 24, 25]. This case study is particularly appropriate because the utility of hash functions depends on the layout of data in memory. This layout depends on the linking mode as well as on address translation. E.g. a large stride crossing page boundaries in the virtual address space may not be a stride pattern any more in the physical address space.

The remainder of this paper is organized as follows. Related work is discussed first in Section 2. In Section 3 we discuss hash functions, the case study used throughout this paper. Section 4 discusses the experimental setup and Section 5 presents and analyzes simulation results. Section 6 concludes the paper.

2 Related Work

Validating microprocessor simulators is impossible without having a real machine available to validate the simulator against [3, 7, 17]. Simulators can be inaccurate because of three types of errors [3]. Modeling errors occur when the desired functionality is correctly understood, but the simulator is coded wrongly. Specification errors occur when the functionality is not properly understood and a wrong model is implemented. Abstraction errors occur when the functionality is modeled at the wrong level of detail. Abstraction errors are purposefully introduced, e.g., ignoring unimportant effects increases simulation speed while introducing only a very small error. The effects described in this paper are abstraction errors.

When validating microprocessors, it has been possible to obtain very accurate models for the processor core, but the accuracy of the memory hierarchy models remained much lower. Desikan, Burger and Keckler [7] used micro-benchmarks to validate a simulator for the Alpha 21264 processor. They could identify many performance bugs, except for the memory system. In the end, they simply tuned the parameters of the memory system to minimize the error, averaged over their micro-benchmarks.

Gibson *et. al.* [8] compare the source and magnitude of errors in architectural simulators with different levels of abstraction. They found that simulators that did not model the operating system introduced large errors in the performance estimate. The common belief that the operating system may be ignored when the benchmarks make few system calls (as is the case in the SPLASH-2 benchmarks used by Gibson *et. al.* [8] and for the benchmarks used in this study) was shown to be invalid when the programs thrash the TLB. In this case, accurate performance estimates require (i) modeling the TLB and (ii) accurately modeling the latency of a TLB miss. It is shown in this paper that another source of errors is related to the mapping of virtual to physical addresses. This mapping was correctly modeled by Gibson *et. al.* [8].

Cain *et. al.* [4] show that, even for the computation-intensive SPEC2000 benchmarks, the operating system has a notable impact on the number of level-2 cache misses. They attribute this effect to the presence of cache control instructions such as the dcbz instruction in the PowerPC architecture. This instruction installs a cache block without accessing the next level of cache when it is known that the

cache block does not contain valid data, e.g., when the page is freshly allocated. Cases like these remove many cache misses in user mode code.

Chen and Bershad [5] compare a deterministic mapping of virtual addresses to a random mapping. The deterministic mapping maps consecutive virtual pages to consecutive physical pages. It can perform significantly worse than a truly random mapping.

3. Case Study: Hashing in the Level-2 Cache

The basic idea behind hashing is best explained using the example of a strided array. It is known that, for certain values of the stride, the elements of the array are mapped only to a subset of the sets of the cache, causing a large number of conflict misses. This behavior is removed when the stride is relatively prime with the number of sets (times the block size). A situation that works for all strides occurs when the number of sets is a prime. Fast implementations of division by a prime exist for certain primes [13, 27].

The complex division by primes is not necessary to obtain conflict-free hashing for stride patterns. Rau [19] showed that XOR-based hash functions can provide conflict-free mapping for a large number of strides. We evaluate a XOR-based hash function that computes the bitwise XOR of two equal-length slices of the block number.

We selected the example of the hash function to test the importance of correctly simulating address translation and the impact of the linking mode because both these parameters impact the data layout. When the data layout is modified, then the hash function may become less (or more) effective, such that evaluating the utility of the hash function is a good test to decide on the level of detail required in the simulation of level-2 caches.

4 Experimental Setup

4.1 Simulation Environment

Our simulation environment requires access to virtual-to-physical address translations and it needs to simulate dynamically linked programs. Furthermore, it is necessary to model the variability present in the virtual address translations. The first two requirements are fulfilled by full-system simulators such as SimOS [20] and SimICS [16]. However, these simulators do not reflect the variability present in address translations. They are typically used by taking a checkpoint of the system after the operating system has booted and initial system activity has finished. Then, each benchmark is simulated starting from the same checkpoint. This method is deterministic, i.e. each run produces the same performance estimate. Variability in the address translations can only be enforced by taking checkpoints at different moments, hoping to obtain sufficiently different initial configurations of the page tables such that the correct variability is measured. It is, however, not clear how to properly load the system in order to obtain random initial configurations.

We follow a different approach. We constructed a direct-execution simulation environment consisting of four components: (1) a running system, (2) the benchmark, running natively, (3) the DIOTA dynamic instrumentation toolkit and (4) a simulator of a 3-level memory hierarchy.

DIOTA[1] is a dynamic instrumentation toolkit that instruments a program on-the-fly [15]. DIOTA allows the user to insert calls to instrumentation functions at basic block boundaries, when memory accesses occur, etc. These instrumentation routines receive information about the current value of the program counter, the memory addresses that are accessed, etc. DIOTA may incur some perturbations as it requires memory to operate. These perturbations are minimized by allocating memory from a different heap than the program. Furthermore, they are constant across runs of a program when the same code path is followed.

For dynamically linked programs, DIOTA and the memory simulator are simply loaded as shared libraries. To instrument statically linked programs, DIOTA provides its own loader that attaches these shared libraries to the statically linked program.

DIOTA allows us to extract a memory access stream. However, as DIOTA operates in the address space of the instrumented program, only virtual addresses are available. The memory simulator requires physical addresses to index the level-2 cache. These addresses are obtained by querying the operating system. Hereto, small modifications were applied to the Linux kernel. We added a function to translate a virtual address to a physical address for a desired process. This function was made accessible to user programs by means of a device driver. Address translations can now be obtained by performing an `ioctl()` call on the device driver. The device driver solution was chosen as it is a simple way to call the kernel and return an argument.

Address translations are requested only when the level-2 cache is accessed. Since `ioctl()` calls are fairly slow (task switches are unavoidable) and since swapping is not very frequent, the address translations are cached by the memory simulator. To allow the simulator to track real swap activity, the cache of address translations is flushed once every 1 million executed basic blocks, but the old translations are remembered. When a new address translation is requested (and misses the cache), the new physical address is compared to the old value. If it changed, it is assumed that the page must have been swapped out and back in, so the page is flushed from all caches. Page changes occur

[1] `http://www.elis.UGent.be/diota/`

Table 1. List of benchmarks used in the study.

Name	Suite	Input
applu	SPECfp2000	reference
equake	SPECfp2000	reference
mgrid	SPECfp2000	reference
swim	SPECfp2000	reference
bzip2	SPECint2000	input.program 58
gap	SPECint2000	reference
mcf	SPECfp2000	reference
parser	SPECint2000	reference
bt	NPB	class A
cg	NPB	class S
ft	NPB	class W
lu	NPB	class W
mg	NPB	class W
sp	NPB	class A
mst	Olden	1024 1
tree	treecode	3 8192
sparse	SparseBench	20x20 irregular patterns

rarely in our simulations.

The experiments are performed on a 2 GHz Intel Pentium 4 processor with 512 KiB level-2 cache and 1 GiB of main memory. The operating system is Fedora Core 1 running the Linux kernel version 2.6.4. The system is experiencing a mild load during the measurements, i.e. text editing, reading e-mail, etc.

4.2 Benchmarks

Seventeen benchmarks are taken from the SPEC2000 suite, the NAS Parallel Benchmarks[2] (NPB, serial version), the Olden benchmarks[3], treecode[4] and SparseBench[5]. (Table 1). For the SPEC benchmarks reference inputs are used. Inputs for the NPB benchmarks are selected in order to obtain varying results.

The benchmarks are compiled using the Intel C/C++ compiler for IA-32, version 8.0 and the Intel Fortran Compiler for IA-32, version 8.0. The compilers are run with the flags `-O3 -ipo`. The `-static` flag is added to produce statically linked versions of the code. Each benchmark is simulated for at most 500 million basic blocks. Although it is preferable to simulate a properly chosen slice of each program [21], there was no easy way to do this since we want to compare static and dynamic linking (each has different instruction counts).

[2] `http://www.nas.nasa.gov/Software/NPB`.
[3] `http://www.cs.wisc.edu/~amir/olden.plain.dist.tgz`.
[4] `http://www.ifa.hawaii.edu/~barnes/treecode/treeguide.html`.
[5] `http://www.netlib.org/benchmark/sparsebench/`.

5 Evaluation

5.1 Measurement Results

We measure the average memory access time (AMAT) of a 3-level memory hierarchy. The level-1 instruction and data caches are each 16KiB large, 2-way set-associative and have 32 byte blocks. The level-2 cache is 128 KiB large, 2-way set-associative and has 64 byte blocks. The main memory is 1 GiB large. The latencies of the caches are 1 cycle for the level-1 caches and 25 cycles for the level-2 cache. Main memory latency equals 581 cycles. These numbers were measured for a 2 GHz Pentium 4 processor.

We set up a three-factor full factorial design with replications [12] to determine the effect of virtual address translation and linking mode on the utility of different types of hash functions. The controlled parameters are (i) address translation (V=virtual, P=physical), (ii) linking mode (S=static, D=dynamic) and (iii) the hash function (B=base, M=prime hashing, X=XOR-based hashing). The prime hashing function uses the prime 1021 [13] and the XOR-based hash function computes the XOR of the lowest 10 bits of the block number with the next 10 bits. In order to measure the variance on the performance metric, each benchmark is simulated 10 times in each configuration.

Figure 1 and Figure 2 show the mean over 10 runs. The error bars have a length of one standard deviation. Note that the AMAT numbers are sometimes large. This is due to the high latency of main memory and the absence of latency hiding in our models. The conditions are indicated by three letters (A/H/L), standing for address translation, hash function and linking mode.

The results show that approximating the physical address with the virtual address is not accurate at all. Performance can be underestimated by as much as 24% for gap and 21% for tree or it can be overestimated by about 24% for bt and 19% for ft. Thus, ignoring the virtual to physical address translation performed by the operating system leads to large errors in estimating memory system performance.

It is also clear that the way the program is linked has little effect on performance for the majority of benchmarks. In some cases, however, there is a large difference between dynamic and static linking, e.g. tree has around 12% higher performance with static linking, while ft has about 20% lower performance.

It is hard to judge which conditions lead to a lower AMAT when the confidence intervals overlap. To compare these, we need statistical methods, as explained next.

5.2 Statistical Analysis

The Kruskal-Wallis hypothesis test [6] determines if the means of two samples are equal. This test is non-

(a) applu

(b) equake

(c) mgrid

(d) swim

(e) bzip2

(f) gap

(g) mcf

(h) parser

Figure 1. Spread of average memory access time under each of the 12 conditions.

parametric, i.e., it does not assume that the data is sampled from a known distribution.[6]

The Kruskal-Wallis test chooses one of two hypotheses: the null hypothesis (H_0) stating that the means are equal and the alternative hypothesis (H_A) stating that the means are different. The null hypothesis is tested by computing a test statistic from the measured data. The test statistic is typically called T. It is a random variable because it is computed from random data. The distribution of T is known when the null hypothesis is true. When the null hypothesis is true, then T will take on a likely value. However, when T has a value that is unlikely given its distribution under the null hypothesis, then the null hypothesis is probably wrong. The null hypothesis is rejected with a significance level $alpha$ if the test statistic falls in the region of the α least likely values (see Figure 3). Other values of the test statistic lead to acceptance of the null hypothesis.

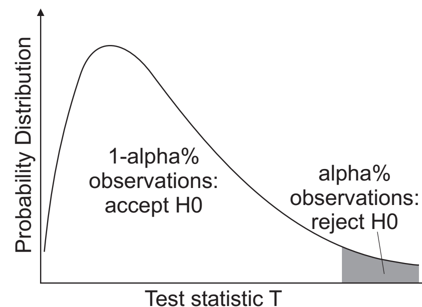
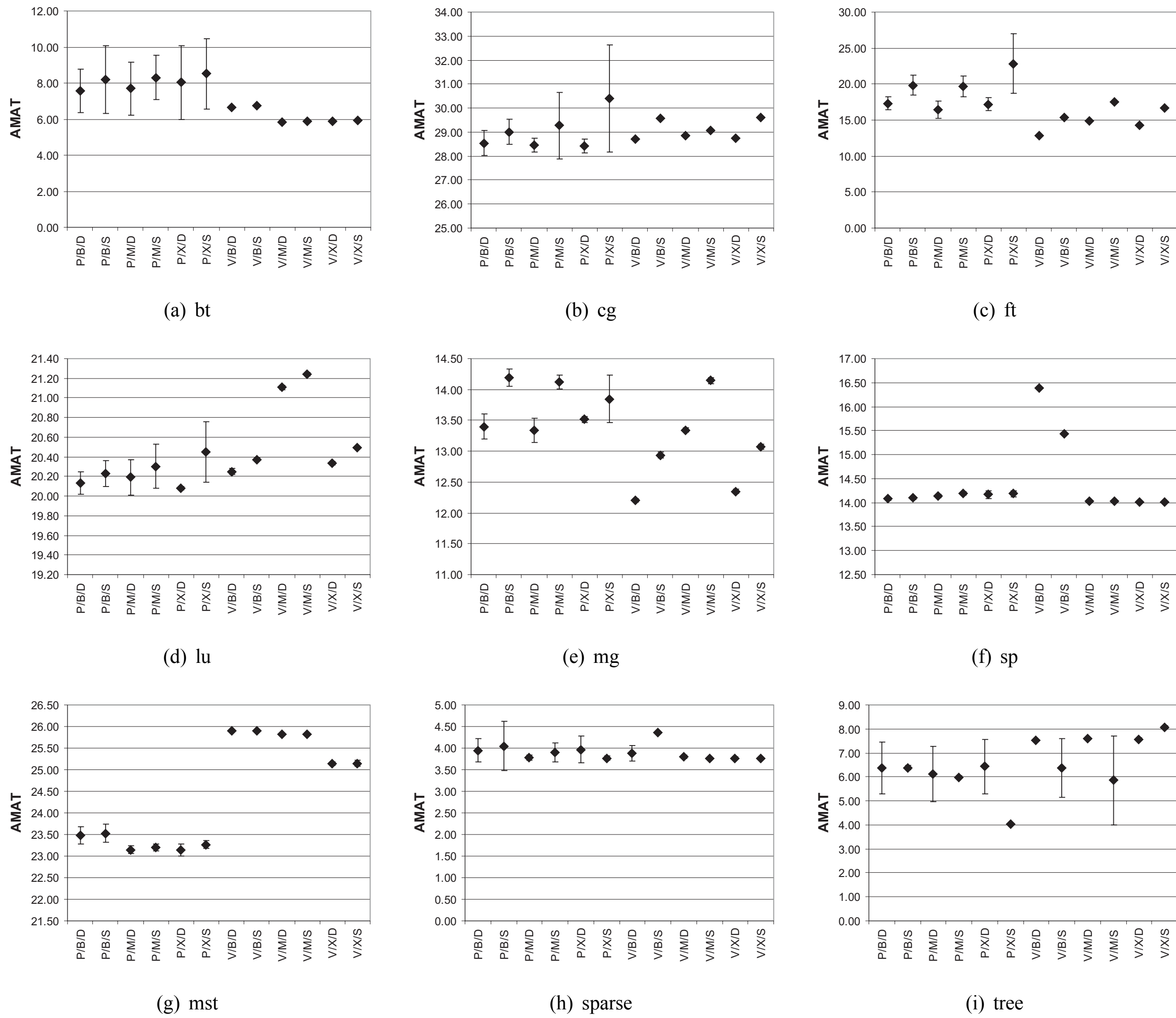


Figure 3. Illustration of the statistic test of equality of means.

[6]The t-test makes the assumption that the population distribution is gaussian. We cannot use the t-test as this assumption is absolutely not valid for our data.



(a) bt (b) cg (c) ft (d) lu (e) mg (f) sp (g) mst (h) sparse (i) tree

Figure 2. Continuation of Figure 1

To facilitate the interpretation of the test it is common to compute a p-value indicating how likely it is that the null hypothesis should be rejected given the measurement data. The p-value is the smallest significance level at which the null hypothesis would be rejected [6]. The null hypothesis is rejected if the p-value is less than or equal to α.

We use a significance level of $\alpha = 5\%$. If this level of significance is achieved, then the difference between the means of the compared samples is statistically significant.[7]

Using the Kruskal-Wallis test, we evaluate whether the choice of linking mode is statistically significant (Table 2). We divide all measurements shown in the previous section in two groups: those measurements where the programs are linked statically and those measurements where dynamic linking is used. Within each group, there is variation in hash function and modeling of address translation.

The p-value is less than 5% for 7 of the 17 benchmarks. Thus, the null hypothesis is rejected for equake, gap, mcf, parser, bt, cg, ft, lu, mg and tree, meaning that the linking mode has a significant impact on performance. Inspection of the means (Figures 1 and 2) shows that the differences are significant performance-wise for only a few benchmarks: equake (9%), ft (20%), mg (5%) and tree (12%). It is not possible to determine on statistical grounds whether static or dynamic linking results in lower miss rates for the other 7 benchmarks.

Next, we analyze the impact of address translation on performance. This impact is not statistically significant for only one program: swim. This is due to the setup of our

[7] A statistically significant performance difference is not the same as a "significant" performance difference. A statistically significant performance difference has been proven using statistic means, but may be very small. A "significant" performance difference is large, but not necessarily statistically proven.

Table 2. Analysis of the effect of linking. Column "df" shows the degrees of freedom of the test statistic "T". The last column shows what type of linking results in higher performance. The difference is significant only if the p-value is less than 5%.

Program	T	df	p-value	Best
applu	0.0069	1	0.93	dynamic
equake	63.35	1	1.73e-15	static
mgrid	0.067	1	0.80	dynamic
swim	0.73	1	0.40	static
bzip2	0.015	1	0.90	dynamic
gap	21.12	1	4.30e-06	static
mcf	7.02	1	0.0081	dynamic
parser	80.43	1	3.0095e-19	dynamic
bt	5.02	1	0.025	dynamic
cg	44.72	1	2.27e-11	dynamic
ft	40.32	1	2.15e-10	dynamic
lu	13.38	1	0.00025	dynamic
mg	25.81	1	3.76e-07	dynamic
sp	0.070	1	0.79	static
mst	0.18	1	0.67	dynamic
sparse	0.00065	1	0.98	dynamic
tree	4.65	1	0.031	static

Table 3. Analysis of the effect of address translation. Column "df" shows the degrees of freedom of the test statistic "T". The last column shows whether physical addresses or virtual addresses result in higher performance. The difference is significant only if the p-value is less than 5%.

Program	T	df	p-value	Best
applu	83.69	1	5.80e-20	physical
equake	32.44	1	1.23e-08	physical
mgrid	19.99	1	7.79e-06	virtual
swim	2.78	1	0.095	physical
bzip2	84.53	1	3.79e-20	physical
gap	84.52	1	3.79e-20	physical
mcf	84.52	1	3.80e-20	virtual
parser	15.83	1	6.94e-05	physical
bt	69.22	1	8.82e-17	virtual
cg	18.29	1	1.90e-05	physical
ft	51.95	1	5.71e-13	virtual
lu	50.49	1	1.20e-12	physical
mg	41.89	1	9.64e-11	virtual
sp	8.94	1	0.0028	physical
mst	84.52	1	3.80e-20	physical
sparse	4.65	1	0.031	virtual
tree-8192	28.16	1	1.12e-07	physical

experiment: we compare all results using address translation to all results without address translation. These two sets of results have the same average performance. There is a performance impact of the hash function when the level-2 cache is indexed with virtual addresses, but this effect does not exist when the simulation is correctly performed with physical addresses. This points to conflict misses between pages that are easily removed by not allocating virtual pages to consecutive addresses.

For 10 of the 16 benchmarks, indexing the level-2 cache with physical addresses underestimates the cache miss rate, resulting in large performance differences for the majority of benchmarks, e.g., 24% for gap and bt, 22% for tree and 19% for ft.

5.3 The Impact on Drawing Wrong Conclusions

Using inaccurate simulators creates the possibility of drawing wrong conclusions from seemingly correct experiments [3, 8]. Inaccurate simulators model some bottlenecks in the processor in the wrong way. When adding a feature to a processor, it is possible that the bottlenecks change, hiding one bottleneck and exposing another one. When these bottlenecks are modeled incorrectly, the utility of the feature may be over- or underestimated.

Furthermore, Alameldeen and Wood [1] have shown that it is also possible to draw wrong conclusions with a correct simulator when the variability in the measurements is not properly taken into account. When only one run of the simulator is performed under each condition, then it is possible that, due the variability in performance, the better condition is perceived worse.

The probability of drawing a wrong conclusion is measured by the *wrong conclusion ratio* (WCR), comparing two alternatives (in this case, hash functions) in the following way. First, it is decided whether hash function A is better than hash function B in the more realistic situation, i.e., indexing the level-2 cache with physical addresses and using dynamically linked benchmarks. This decision is made using the average performance over the 10 measurements. It serves as the correct outcome that we want to obtain using experiments. Second, we assume that the experiments are performed under the common condition of indexing the level-2 cache with virtual addresses and using statically linked benchmarks. We also assume that variability is ignored and only one experiment is performed. In this situation, the probability of drawing a wrong conclusion equals the number of pairs of simulations (one simulation to evaluate hash function A and one simulation to evaluate hash function B) in which the comparison of A to B does not agree with the correct outcome [1].

Two values are common for the WCR: 0% and 100%. A WCR of 0% indicates that the approximation to the simulation environment of ignoring address translation and linking

Table 4. Wrong Conclusion Ratio. Each comparison shows two columns: the configuration that has the lowest AMAT on average and the percentage of experiment pairs that lead to the wrong conclusion. In column "A–B", the $<$ sign indicates that the correct evaluation is that "A" has lower AMAT than "B", while $>$ means the opposite.

	Base–Prime		Base–XOR		Prime–XOR	
	True	WCR	True	WCR	True	WCR
applu	$<$	0%	$<$	0%	$<$	0%
equake	$>$	77%	$>$	97%	$>$	87%
mgrid	$>$	0%	$>$	0%	$<$	0%
swim	$<$	0%	$<$	0%	$<$	0%
bzip2	$>$	0%	$>$	0%	$>$	80%
gap	$>$	57%	$>$	19%	$<$	80%
mcf	$<$	0%	$<$	0%	$<$	0%
parser	$<$	27%	$>$	66%	$>$	48%
bt	$<$	100%	$<$	100%	$<$	0%
cg	$>$	0%	$>$	95%	$>$	100%
ft	$>$	100%	$>$	100%	$<$	100%
lu	$<$	0%	$>$	100%	$>$	0%
mg	$>$	100%	$<$	0%	$<$	100%
sp	$<$	100%	$<$	100%	$<$	100%
mst	$>$	0%	$>$	0%	$<$	100%
sparse	$>$	0%	$<$	100%	$<$	21%
tree	$>$	48%	$<$	0%	$<$	0%
all	$<$	50%	$<$	49%	$<$	50%

mode are not important to evaluate whether one hash function is better than the other. A WCR of 100% indicates that the approximation to the simulation environment is wrong. The WCR is computed when performing three experiments: comparing baseline indexing to prime hashing, comparing the base to XOR-based hashing and comparing prime hashing to XOR-based hashing (Table 4).

Comparing the base to prime hashing shows that, on average, prime hashing leads to worse performance for the simulated benchmarks. In four cases, the comparison is totally wrong. Two cases give the benefit to the baseline (bt and sp) and two cases benefit prime hashing (ft and mg). For 5 more benchmarks, it is possible to draw wrong conclusions due to ignoring variability (equake, gap, parser, tree and also for the average over all benchmarks). Similar remarks can be made for the comparison of the baseline to XOR-based hashing and when comparing prime hashing to XOR-based hashing. Note also that, when comparing the result over all benchmarks, the WCR is 49% or 50%, indicating that the construction of the simulation environment has an important impact on the outcome of the evaluation. As a side-effect of this study, we find that the difference between the three hash functions is not significant for the studied benchmarks.

5.4 Do Hash Functions Decrease Variability?

Some parameters of a simulation are inherently random. Adding a feature to a microprocessor may not significantly alter average performance, but it may reduce the variability on performance. This is also beneficial, as worst case performance improves. In this section, we evaluate whether hash functions decrease variability.

We compare the variance of two random samples using a hypothesis test. Due to space limitations, we only discuss the conclusions of this analysis. We perform pairwise tests between every pair of hash functions, always assuming physical addresses to index the level-2 cache and dynamically linked executables, i.e. the P/../S configurations. There is a statistically significant difference in variance between the baseline hash function and prime hashing for only two benchmarks (mst and sparse). In both cases, prime hashing reduces variance. XOR-based hashing reduces variance compared to baseline for all benchmarks. This difference is significant for 4 benchmarks (swim, lu, mg and sp). The third test, comparing prime hashing to XOR-based hashing, gives results consistent with the other tests. Note that the reduction in variance can occur when the hash function also improves performance (mst, sparse and lu), but also when performance is decreased (swim, mg and sp).

6 Conclusion

We have investigated the importance of correctly modeling virtual-to-physical address translation and the linking mode of programs (static versus dynamic linking) on simulation accuracy. We have shown that performance estimates can be wrong by as much as 24% when neglecting to model virtual address translation. Using static linking usually introduces negligible errors but for some benchmarks the errors can be up to 20%. In both cases, performance can be either overestimated or underestimated.

Furthermore, correctly modeling these two effects introduces performance variability: virtual address translation varies from run to run, causing a difference in access patterns to physically indexed caches (e.g., the level-2 cache). These differences translate into variations in the miss rate. We show that, in order to correctly evaluate the impact of optimizations to the level-2 cache, it is important to take this variability into account. Evaluating the utility of hash functions in the level-2 cache can easily lead to wrong conclusions when the simulation environment does not properly model virtual address translation or does not allow dynamically linked programs.

Furthermore, based on the data that we gathered during our experiments, we show that hash functions in the level-2 cache are not useful to improve performance, when averaged over a range of benchmarks. However, the variability

in performance is reduced when adding a XOR-based hash function to the level-2 cache.

Acknowledgements

We are indebted to Jonas Maebe for support on DIOTA and to Michiel Ronsse for his help on the device driver. Hans Vandierendonck is a post-doctoral researcher with the Fund for Scientific Research-Flanders (FWO-Vlaanderen). This research is sponsored by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT) and Ghent University.

References

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 7–18, Feb. 2003.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 36(2):59–67, Feb. 2003.

[3] B. Black and J. P. Shen. Calibration of microprocessor performance models. *IEEE Computer*, 31(5):59–65, May 1998.

[4] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-5), in conjunction with HPCA-8*, Feb. 2002.

[5] J. B. Chen and B. N. Berhsad. The impact of operating system structure on memory system performance. In *Proceedings of the fourteenth ACM symposium on Operating systems principles SOSP '93*, pages 120–133, 12 1993.

[6] W. J. Conover. *Practical Non-Parametric Statistics*. John Wiley & Sons, 1999.

[7] R. Desikan, D. Burger, and S. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, June 2001.

[8] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, Nov. 2000.

[9] N. Gloy, C. Young, J. B. Chen, and M. D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 12–21, May 1996.

[10] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *ICS'97. Proceedings of the 1997 International Conference on Supercomputing*, pages 76–83, July 1997.

[11] J. L. Hennessy and D. A. Patterson. *Computer architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[12] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[13] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 288–299, Feb. 2004.

[14] T. Li, L. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio. Understanding and improving operating system effects in control flow prediction. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 68–80, Oct. 2002.

[15] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Workshop on Binary Translation. In conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[16] P. S. Magnusson, F. Dahlgren, H. Grahn, and et al. SimICS/sun4m: A virtual workstation. In *Proceedings of the Usenix Annual Technical Conference*, June 1998.

[17] D. Ofelt and J. L. Hennessy. Efficient performance prediction for modern microprocessors. In *SIGMETRICS'2000. Proceedings of the 2000 ACM Conference on Measurement and Modeling of Computer Systems*, pages 229–239, June 2000.

[18] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual. Technical Report Technical Report 9705, Dept. of Electrical and Computer Engineering, Rice University, July 1997.

[19] B. R. Rau. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 74–83, May 1991.

[20] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, Winter 1995.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[22] D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, Nov. 1987.

[23] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[24] H. Vandierendonck and K. De Bosschere. XOR-based hash functions. *IEEE Transactions on Computers*, 54(7):800–812, Sept. 2005.

[25] H. Vandierendonck, P. Manet, and J.-D. Legat. Application-specific reconfigurable XOR-indexing to eliminate cache conflict misses. In *Design, Automation and Test Europe*, pages 357–362, mar 2006.

[26] J. E. Veenstra and R. J. Fowler. MINT tutorial and user manual. Technical Report Technical Report 452, Computer Science Department, University of Rochester, June 1993.

[27] Q. Yang and W. LiPing. A novel cache design for vector processing. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 362–371, May 1992.