Extrinsic and Intrinsic Text Cloning

Marios Kleanthous, Yiannakis Sazeides, Marios D. Dikaiakos

Extrinsic and Intrinsic Text Cloning

Marios Kleanthous, Yiannakis Sazeides and Marios D. Dikaiakos
Department of Computer Science, University of Cyprus
75 Kallipoleos Street, P.O.Box.20537, CY-1678 Nicosia, Cyprus
{mklean,yanos,mdd}@cs.ucy.ac.cy

Abstract

Text Cloning occurs when a processor is storing in one or more levels of its cache hierarchy the same text multiple times. There are several causes of Text Cloning and we classify them either as Extrinsic or Intrinsic.

Extrinsic Text Cloning can happen due to user and software practices, or middleware policies, which result into making multiple copies of a binary and concurrently executing the multiple copies on the same processor.

Intrinsic Text Cloning can happen when an instruction cache is Virtually Indexed/Virtually Tagged and the process identifier is included in the tag. A simultaneous multithreaded processor, that employs such cache, will map the text of concurrent processes of the same binary to different instruction cache space due to their distinct process identifier.

Text cloning can be wasteful to performance, especially for simultaneous multithreaded processors, because concurrent processes compete for cache space to store the same instruction blocks. Experimental results on simultaneous multithreaded processors indicate that the performance overhead of this type of undesirable cloning is significant. Theses findings call for OS and/or architectural support to reduce or eliminate Text Cloning.

1 Introduction

Power constraints and diminishing returns from increasing the issue width on superscalar processors have lead to the emergence of general-purpose single-chip multi-core processors. Furthermore, with continuous technology miniaturization more and more cores are integrated on-chip. Cores are typically multithreaded [27, 25] to leverage unutilized core resources, due to a stalled or low performing thread, to execute concurrently multiple threads in the same processor. Niagara2 [19] contains 8 cores each supporting 8 threads, Intel's i7 [6] contains a quad core each 2-way simultaneous multithreaded (SMT), and Power7 [22] contains 8 cores each 4-way SMT.

The combination of multi-cores and multi-threading is effective in improving processor utilization as long as the memory hierarchy can satisfy all running threads instructions and data needs. Consequently, modern processors devote a large fraction of their real estate for the cache hierarchy and numerous research studies are conducted on how to efficiently share the cache hierarchy among concurrent on-chip threads [3, 7, 20].

In this work we identify Text Cloning as a potential inefficiency in the cache hierarchy of modern multi-core processors. Text Cloning occurs when a processor is storing at one or more levels of its cache hierarchy the same text multiple times. Text cloning can be wasteful to performance, especially for SMT cores, because processes compete for cache space to store the same instruction blocks at the same time. There are several causes of text cloning and we divide them into Extrinsic and Intrinsic.

Extrinsic Text Cloning can happen when a user, many users, or middleware, copy a binary and concurrently execute the multiple copies on the same processor. The Operating System is unable to detect that these binaries are identical and will map them during execution in different physical address space, therefore, creating unnecessary pressure at all cache levels. Such a scenario is very common in Grid Computing job flow where the binary of each submitted job is copied in a temporary directory, a sandbox, with all its inputs and data.

Intrinsic Text Cloning can happen when an instruction cache is Virtually Indexed/Virtually Tagged and the process identifier (PID) is included in the tag. A simultaneous multithreaded processor , that uses such cache, will map the text of concurrent processes of the same binary to different instruction cache space due to their distinct process identifier. A Virtually Indexed/Virtually Tagged instruction cache is found in the Intel's hyperthreaded (SMT) Netburst microarchitecture [13].

This paper identifies and explains the causes of Text Cloning both, Extrinsic and Intrinsic, and demonstrates experimentally, on real and simulated SMT hardware, the significant performance implications of Text Cloning. The paper discusses ways to mitigate the effects of Text Cloning and shows the potential of a hardware-based approach to identify and eliminate it.

The rest of the paper is organized as follows: Section 2 discusses the various causes of Extrinsic and Intrinsic Text Cloning, demonstrates the sensitivity of real hardware performance to text cloning and discusses possible ways to mitigate its detrimental effects. Sections 3 describes in detail the problem of Text Cloning in Grid Computing Systems. Section 4 presents simulation based experimental analysis that underlines the importance of eliminating text cloning and demonstrates that a hardware based scheme can effectively identify text cloning and eliminate it. Section 5 presents related work on cache duplication and, finally, in Section 6 we conclude.

2 Text Cloning: Causes, Implications, Remedies

This section introduces Extrinsic and Intrinsic Text Cloning through discussion about when it can occur, how much it hurts performance and possible methods to avoid it.

2.1 Extrinsic Text Cloning

Extrinsic Text Cloning (ETC) can happen due to user and software practices that result in the execution of multiple copies of the same binary on the same processor. The Operating System is unable to understand that these binaries are clones and will map them in different physical address spaces. Consequently, each process is associated to a different text segment and will eventually create duplication in shared caches of the processor.

The ETC is common within Grid Computing Systems [10] due to Grid's distributed file system and the middle-ware design. In particular, typical Grid job flow requires the binary of each submitted job to be copied in a temporary directory, a sandbox, with all its inputs and data. Although Grid computing consists of a large number of computing nodes and provides high throughput, its efficiency is highly dependent on the middle-ware that schedules and submits the different jobs to computing nodes. In the case that two or more jobs, that use the same binary, are submitted to the same multicore or SMT computing node the middle-ware, or even the OS in the Grid computing node itself, is unaware of this duplication.

Another emerging case of ETC is due to virtualized cloud computing where multiple users can run local copies of the same applications that happen to execute on the same physical processor [18].

Furthermore, ETC can happen when an application contains self-modifying code routines. When a process, that shares its physical address space with other processes, self modifies its code then the memory page that contains the modified code has to be copied in different address. This will result to duplicated blocks that were contained in the copied memory page but remain unaffected from the code self-modifying routine [2].

Finally, a common habit among users is to keep their own copies of same applications in their home directories. This might lead in ETC when two users are logged in the same machine and run the same application, using their own copy.

2.2 Intrinsic Text Cloning

Intrinsic Text Cloning (ITC) is specific to VIVT instruction caches. A VIVT cache uses the Virtual Address to tag match a block. In the case of a shared VIVT cache the tag also contains the PID of the process to avoid homonym problems. However, each instance of the application will have different PID and this will create synonyms [23] in the instruction cache. ITC is equivalent to the occurrence of synonyms in an instruction cache.

VIVT caches are used for L1 Instruction caches to have lower access latency and lower energy per access by avoiding ITLB translations on every cache access. Cloning in IL1 caches only occurs when the tag of the Virtually Tagged (VT) caches includes also the PID. Single thread cores do not require keeping the process ID in the tag unless they want to avoid cache flashing after each context switch. On the other hand, for an SMT processor, the PID is essential in the tag of a VT cache because multiple threads co-exist in the cache at the same time.

The ITC can happen either when we run multiple copies of the same binary or multiple instances of the same binary. For the first scenario, the reasons are the same as those discussed in Section 2.1. The second scenario, multiple instances of the same binary, is very common when running the same application with different inputs, or using applications that by default create a different process for each instance due to lack of multithreading support or other programming reasons. For example, versions of Microsoft Excel and Internet Explorer create a distinct instance each time they are invoked.

Another possible cause of ITC is the service daemons running on servers. Not all of these applications are multithread, and create a different process each time a user request the service. A very common category of services that spawns multiple processes are the kernel services.

2.3 How important is ETC and ITC

This Section uses two real processors with 2-way SMT cores, the Intel Pentium 4 (P4) [17] with VIVT 12KB Trace Cache and the Intel i7 [6] with a VIPT 32KB IL1 cache to measure the performance impact of Text Cloning in IL1 cache. We used a synthetic benchmark (see APPENDIX) that exercises the instruction cache by executing a large basic block of calculations for different basic block sizes. The benchmark has minimal data requirements, only few initial capacity misses, effectively no-conditional branches and several random indirect unconditional branches to measure only the impact of the instruction references on performance.

We measure the implications of ETC and ITC by performing two experiments for each processor. First, two instances of the same binary are executed in parallel. The OS is aware that both processes refer to the same binary and it will load the text only once in the physical address space but it will create two different virtual address spaces, one for each process. This causes ITC only in the P4 with VIVT caches since the address mapping of the threads in the i7 VIPT cache will be the same. For the second experiment, two copies of the same binary are run again in parallel for the SMT execution. This causes the two processes to be mapped in different physical address spaces and as a result different virtual address spaces. This manifests into Text Cloning both for P4 and i7 caches in all levels of the cache hierarchy.

For both experiments, the two processes are forced to run on the same logical core using the taskset command. In this way the two processes will be executed in parallel using one SMT core and share the same IL1 cache.

2.3.1 Intel Pentium 4 with a VIVT IL1

Figure 1 shows the results for the Intel P4. The y-axis of the figure shows the SMT speedup compared to running the two processes back to back. The x-axis shows the static instruction footprint of each process. For the VIVT IL1 cache of P4, running either copies or multiple instances of the same binary does not make any difference. In both cases the two processes will be mapped in different virtual address spaces. The evidence for ETC (two copies) and ITC (same binary) are supported by the behavior from 1KB to 12KB instruction footprint. For this

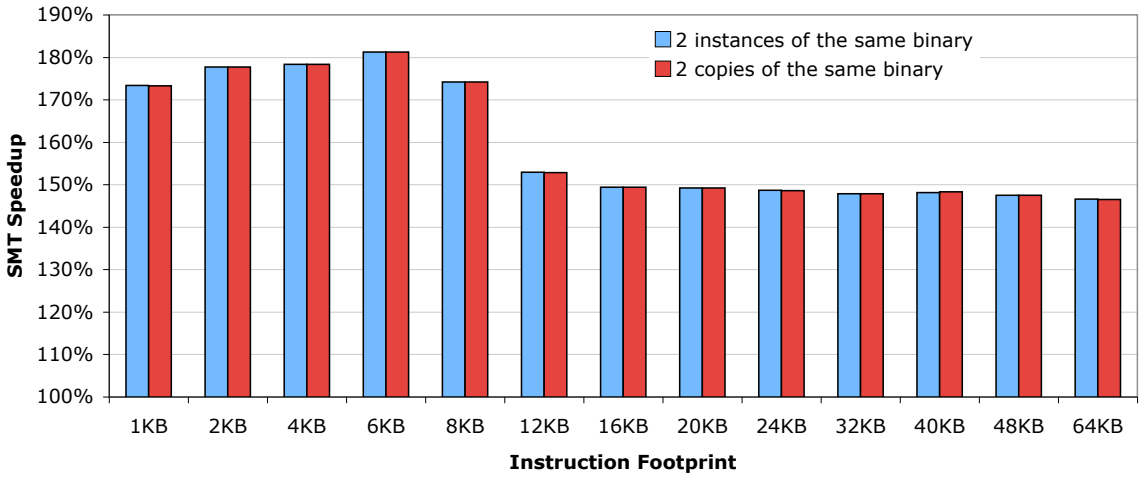

Figure 1: Intrinsic and Extrinsic Text Cloning in Intel Pentium 4

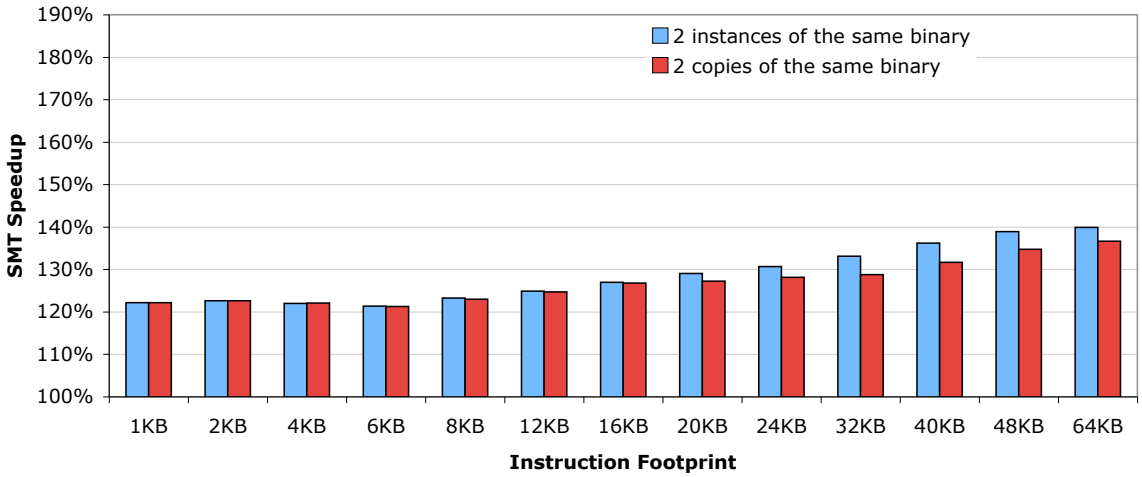

Figure 2: Intrinsic and Extrinsic Text Cloning in Intel i7

sizes the single thread will fit perfectly on the IL1 cache while the SMT executions will suffer with cache misses after the 6KB instruction footprint. In the figure, we can clearly see that the speedup of SMT for both experiments is dropping once the instruction footprint exceeds the 6KB from 80% down to 55% for 12KB.

2.3.2 Intel i7 with a VIPT IL1

Figure 2 shows the effects of running concurrently the same binary and two copies of the binary on an i7. The trends for i7 are clearly different as compared to P4. In particular, comparing the two bars in Figure 2 we observe that when running two different copies of the same binary the SMT speedup is reduced when we go beyond the 16KB instruction footprint because now the combined workload of the two copies occupies 32KB in total which barely fits the i7 32KB IL1 cache. This is clearly due to ETC. On the other hand, the runs with the same binary experience no Text Cloning, as opposed to P4. Specifically, with the 16KB instruction footprint the instructions of both processes are mapped in the same physical space and hence are mapped only once in the VIPT IL1 cache of i7. Comparing Figures 1 and 2 we clearly see that ETC can affect both cores while ITC affects only Pentium 4 that uses a VIVT IL1 cache.

2.4 How to eliminate ETC and ITC

ETC can be avoided if the OS is enhanced with the ability to detect copies of the same binary and map them at the same physical address space. This however can cause security problems since someone can exploit this to inject harmful code in applications that are commonly used among many users.

Another possible solution is to enable the hardware to detect this duplication with hints from the OS or in real time to completely avoid user intervention. At this low level, the detection of cloned text can be more efficient and more secure. Two such mechanisms that have already been proposed are [12, 16] and with certain modifications can be applied to ETC.

More specifically [12] proposed CATCH, a mechanism that dynamically detects and eliminates duplicated instruction sequences, valid blocks, from the IL1 cache. Duplicate instructions sequences can exists because of copy paste programming, macro expansion, function inlining and other compiler and programming optimizations. Mohamood et al. [16] proposed a mechanism to detect DLL sharing between different threads that use the same DLLs. The mechanisms described are based on both VIVT and VIPT caches that are aware of DLL sharing

using a bit in the ITLB table that is set with aid of the Operating System. The mechanisms described can be used to prevent text cloning but we believe that a simpler mechanism may be sufficient because the granularity of duplication is much bigger in the Text Cloning scenario.

ITC can be avoided by using a VIPT IL1 cache. The VIPT cache requires an access to the ITLB on very cache access to translate the Virtual to Physical address. This costs both energy for accessing the ITLB but also performance because even though the Indexing in a VIPT can be done with Virtual address this is not enough to hide the ITLB access and tag matching. This extra translation might increase more than a cycle the IL1 cache access latency. Previous SMT processors, like Intel Pentium 4, kept the L1 Instruction Cache to be VIVT but modern processors, like Intel i7, have a Virtually Indexed/Physically Tagged (VIPT) cache with the extra overhead of the ITLB translation on every IL1 cache access. Therefore, the particular instruction cache configuration may depend on power and performance trade-offs.

Another possible solution for the ITC problem is the hardware mechanisms proposed to detect and eliminate Cache-Content-Duplication dynamically [12, 16]. These mechanism may help eliminate both ETC and ITC.

3 Grid Computing Systems

In this section we will explain in detail how and where Extrinsic Text Cloning manifests in Grid Computing Systems and specifically in EGEE project [1].

3.1 Grid Architecture

Figure 3 shows the basic components of EGEE grid system that uses the gLite middleware to submit, schedule, execute and manage users' jobs. The figure shows that this grid computing systems is composed from four basic elements, (a) the User Interface (UI), (b) the Workload Management System (WMS), (c) the Computing Element (CE) and (d) the Worker Node (WN) [15].

The UI provides the tools for the user to submit or cancel his job and also to retrieve the output result of the submitted job. Once a job is submitted from a UI it arrives to a WMS. The WMS is responsible for the load balancing of the whole grid infrastructure by keeping records of the balance in each cluster and which clusters are available for execution. Once the WMS chooses the cluster to submit a job it sends the job description in a WMS wrapper script to the appropriate CE of the cluster. The CE is responsible for keeping track of the workload in its own cluster and submits jobs to different WNs that belong to the cluster. Finally the WN is running a job resource manager, for EGEE is Torque/PBS, which executes the WMS job wrapper script that setup, download and upload the job's sandbox, execute the job, log and clean up once the job is done.

3.2 Extrinsic Text Cloning in Grid

ETC is caused by the very last stage of the grid job flow, at the WN, where the WMS job wrapper creates a different sandbox for each job. This prevents multiple jobs that run on the same worker node, multicore or SMT, to share their binaries but also provides secure execution of the job.

The architecture of grid is build to provide abstraction in each level but also security for the users to run their job without interfering with each other [10].

This approach provides little or no opportunity to the middleware to optimize job submission and execution to share binaries because there is a high risk of compromising security. For example, even if the WMS component is smart enough to group jobs together that use the same binary and submit them to the same CE it would still need to run in different sandboxes to prevent interference between jobs' inputs and outputs and even malicious activity from other users that may try to exploit this hole.

Accordingly to eliminate ETC in grid computing either the OS running on the worker node or hardware support or a co-design of the two is essential.

For example, a service running in the OS that compares the binaries start executing with binaries already running can be used. This can be done using a table that keeps a content id (e.g. the CRC code) of the text of all running binaries. When a new binary starts executing, its content id is compared with all the running ones and if there is a match the texts are compared for validation. If two texts are identical they can be mapped at the same physical address space. In case of self-modifying code the OS must be aware to split merged texts into different physical address spaces. This technique will require no hardware modifications but requires for the OS to do all the comparisons and monitoring for self-modifying code or another possibly malicious actions from the users.

Another approach is to have a hardware mechanism detecting text cloning. The granularity of duplication can be chosen statically for each set of binaries or it can change dynamically. For example, for two identical binaries only a relation between the PIDs needs to be recorded. On the other hand if two binaries are very similar but not identical, for example an open source simulator that is slightly modified by each user, detection at the granularity of pages or cache blocks is more appropriate. By reducing the detection granularity, the duplication opportunity increases but the number of relations to be recorded increases also. An adaptive mechanism might be useful to keep the cost low when possible by changing the granularity. Smaller granularity also provides duplication detection across very different applications and even within the same binary. Furthermore, detecting self-modifying code and invalidating relations is easier in hardware because it can monitor the instructions that write the text segment.

A more efficient design will be the combination of software and hardware. For example, a co-design where an OS software mechanism provides hints, for the relations and the text cloning granularity, to the hardware mechanism that will validate, create and detect the duplicate relations. The OS has a broader view of the processes running and can detect if two texts are identical, similar or very different. This can help the hardware mechanism to adapt the granularity to detect text duplication. Finally, the hardware can detect self-modifying code and invalidate any relations that become invalid.

Provided that Text Cloning is a frequent phenomenon, future work should evaluate and engineer all these options to determine how to best to detect and eliminate it.

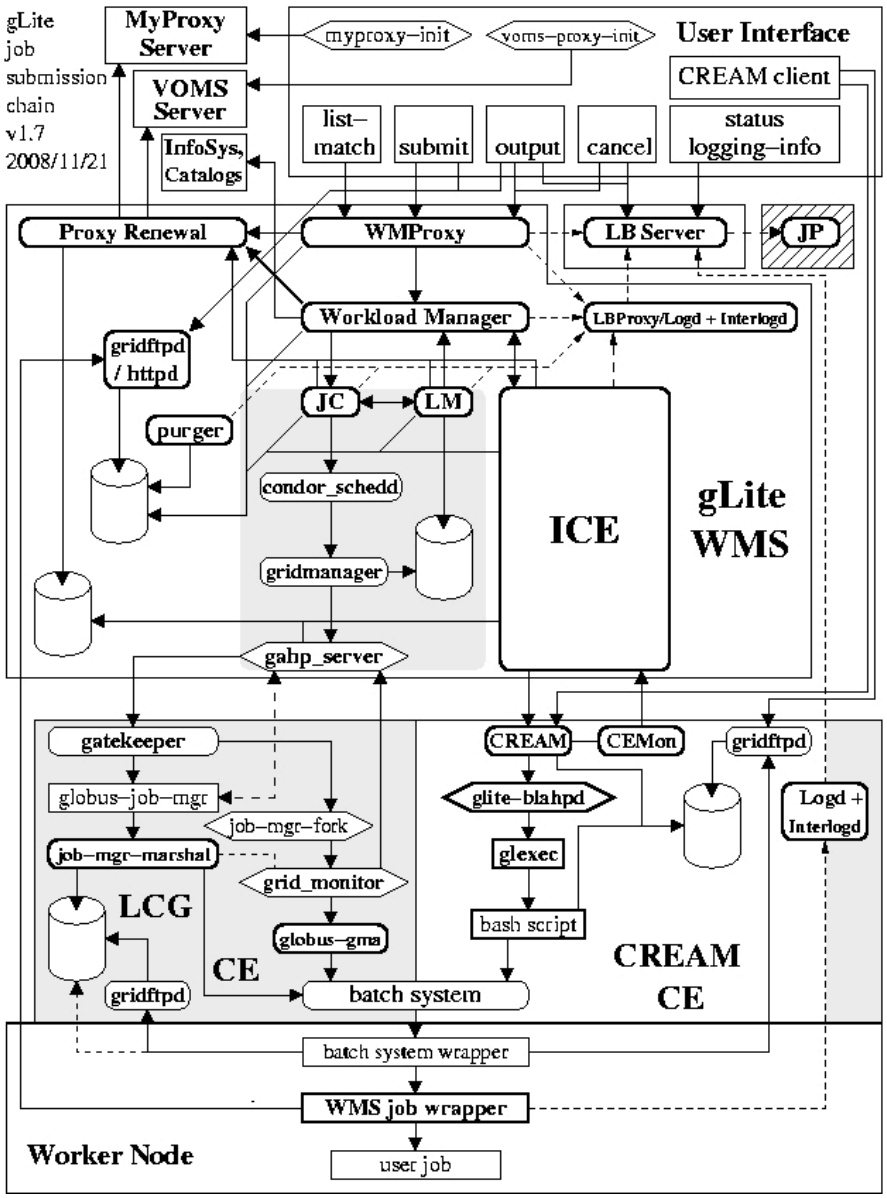

Figure 3: gLite job submission chain (http://web.infn.it/gLiteWMS/index.php/techdoc/howtosandguides)

4 Evaluation using simulation

For simulation evaluation of the effects of text cloning we consider only the scenario where multiple copies of the same binary are executed using a VIPT IL1 cache of an SMT core. This corresponds to a scenario with Extrinsic Text Cloning (ETC).

4.1 Experimental framework

In order to evaluate the performance we have used the SMTSIM simulator [26] with a selection of 7 benchmarks of the SPEC2000 suite. The 7 benchmarks selected where 3 with a large instruction workload, fmad3d, crafty and perl, 2 with a medium instruction workload eon and vortex, and 2 with a small instruction workload load, ammp and lucas. This benchmark selection is done to show the potential performance of ETC for different cases of instruction cache pressure. All benchmarks are run using reference inputs.

Table 1 shows the processor configuration used and Table 2 shows the skipped, executed and shift regions of the benchmarks used. The shift region is the difference in dynamic instructions between the two copies of the binary that are executed simultaneously to avoid overlapping program phases. For these shift regions we have verified that there is no overlapping between the simulated regions of the two copies. The skipped and executed regions were selected using a Simpoint-like tool [21]. Table 2 also includes the misses per 1K instructions for each benchmark for a 16KB L1 instruction cache that are used as criterion for the selection of the benchmarks.

fetch/issue/commit width	4/4/4
INT Issue Queue/FP Issue Queue/ROB	64/64/256
Pipeline Stages	10
L1 instruction cache	VIPT 16KB 8-way 32B/block, 1 cycle
L1 data cache	VIPT 16KB 8-way 32B/block, 1 cycle
L2 unified cache	VIPT 512KB 8-way 32B/block, 20 cycles
Main memory latency	200 cycles

Table 1: Processor Configuration

SPECINT 2000	Skip (10^6)	Execute (10^6)	Shift (10^6)	Misses Per 1K instructions
fma3d	10250	120	500	27.191
crafty	950	240	500	24.841
perlbmk	13800	240	500	21.758
eon	26400	240	500	13.491
vortex	18550	240	500	6.222
ammp	4950	240	500	0.006
lucas	2650	240	500	0.002

Table 2: Simulated benchmarks

4.2 Results

Figure 4 shows the Weighted Speedup [24] normalized to the first bar, which is the performance of 2 instances of the same binary running on an SMT processor. For experiments in Figure 4 all applications are running synchronized, that means they are executing exactly the same program phase. The results show that the performance degradation due to ETC, when running 2 copies of the same binary, is up to 60% for crafty and more than 20% for the other benchmarks. For lucas and ammp that have very little pressure on the instruction cache ETC does not affect the performance.

Figure 5 shows a more common scenario where the two applications running simultaneously are in different program phase, 500 million instructions shift, in their execution. We have verified that none of the applications is overlapping with its copy during the execution. The results here show that the performance degradation is a little less, mainly because by executing a different phase we can avoid some conflict misses. Still the bigger instruction footprint due to ETC can cause 55% slowdown for eon and crafty and about 20% for the other benchmarks. The ammp and lucas are again not affected by ETC due to the very small instruction cache workload.

These results suggest that the use of a hardware mechanism, OS support or a combination of the two will be useful to eliminate the performance degradation due to text cloning. In this work we chose a mechanism, proposed in [12], to show how a hardware mechanism can be used to recover performance loss due to Text Cloning.

Figure 4 shows how CATCH can reduce the overhead of cloning. The third bar shows the performance when two copies of the same binary are executing and CATCH is used to detect and eliminate cloning. We can see that when using CATCH the performance degradation is reduced to 0.07% on average. There is even one case, for vortex, that the performance of CATCH is even better compared to the run where we have executed the same binary twice. This is because CATCH detects duplication not only across different binaries, but also within the same binary and thus improving the performance of the single thread execution.

The results in Figure 5 are similar to 4 but this time we can see that CATCH eliminates completely the cloning overheads on average. We would like to note again that CATCH is not for free and each duplication detection is penalized with one extra cycle that corresponds to an extra cache access to use the duplicated block. The CATCH mechanism is described in detail in [12].

We have used CATCH as a case study to show how a hardware mechanism can be applied to eliminate ETC. The results indicate that an Operating System mechanism or a hardware mechanism that is aware of text cloning can be very useful to improve the performance of modern platforms that suffer from ETC, such as the Grid Computing and Cloud Computing Systems .

5 Related work

Previous work on mitigating code duplication mainly aim to compress the instructions by either profiling the applications or by dynamically detecting and correlating duplicated sequences.

Lefurgy et al. [14] explored the idea of keeping compressed code in instruction memories of embedded processors. Based on static analysis, common sequences of instructions are assigned unique codes. These codes are stored in instruction memory and are expanded to their original form after being read.

Code compaction work has also been dealing with the reduction of static code of a single binary[4, 8, 9]. Code compaction methods are used to reduce the executable code size without a need to decompress the compacted code to execute it.

Harizopoulos and Ailamaki [11] proposed the synchronization of threads in OLTP applications. By synchronizing different threads of the same application to reuse the instructions between them, the total instruction cache footprint of the appli-

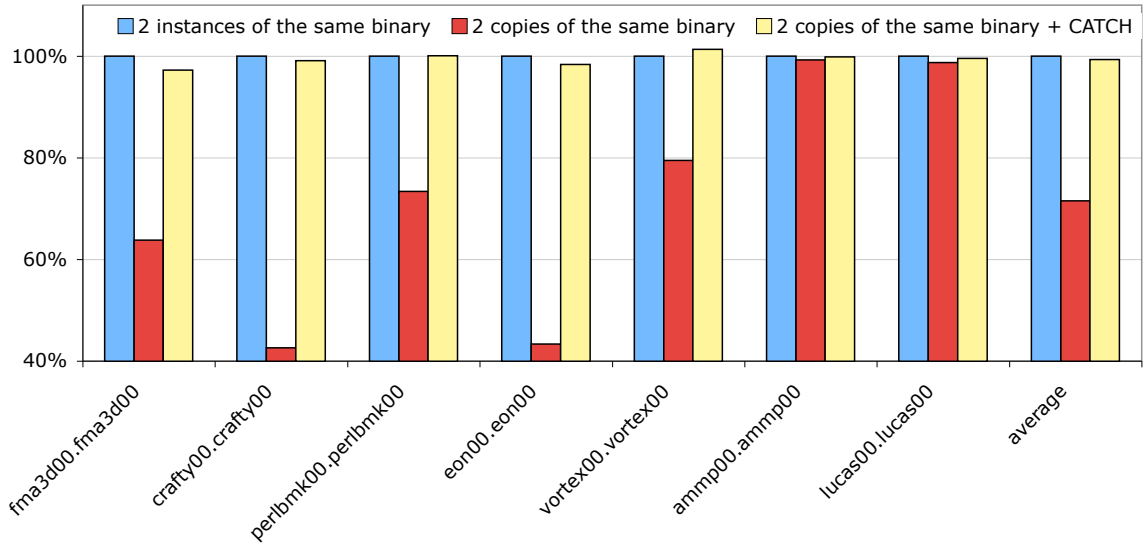

Figure 4: Weighted SpeedUp. Detecting and eliminating ETC with overlapping program phases

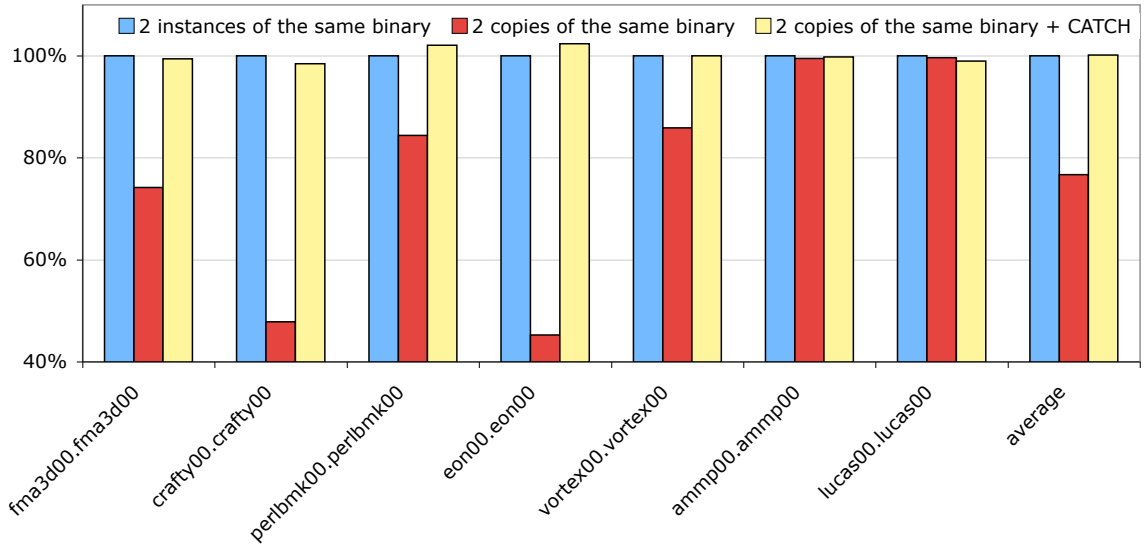

Figure 5: Weighted SpeedUp. Detecting and eliminating ETC with 500 million instructions shift in program phase

cation is reduced.

Biswas et. al. [5] investigate the phenomenon of data similarity in multi-execution programs. They observed that when multiple instances of the same application are running on a multicore sharing the same L2-cache, their data are usually very similar.

All these techniques tried to dynamically detect and exploit duplication at the granularity of cache blocks. In the case of text cloning the duplication can be detected at the granularity of memory pages or even the whole binary, with the help of the Operating System or simple hardware mechanisms.

6 Conclusions

This work analyzes the effects of Extrinsic and Intrinsic Text Cloning (ETC) in caches. Extrinsic text cloning can occur when a binary is copied and executed concurrently multiple times, for example in Grid Computing Systems. In that case the OS is unaware of the Text Cloning and two or more copies of the same binary will be mapped in different physical addresses. Intrinsic Text Cloning (ITC) can occur in the case of Virtually Index/Virtually Tagged caches where the same text segment is mapped in different virtual address spaces.

We evaluate the effects of ETC and ITC, using two SMT Intel processors, P4 and i7 with a synthetic benchmark. The results indicate that the slowdown in execution due to Text Cloning is significant and a mechanism for detecting and eliminating this overhead can be important.

Simulation based evaluation has shown that the performance overheads of ETC can be completely eliminated using a hardware mechanism previously proposed to detect duplication between instruction sequences.

Overall, the analysis in this paper suggests the importance of OS and architectural support to eliminate Text Cloning. As a next step we plan to characterize the Text Cloning in Grid Computing and Cloud Computing frameworks to determine its frequency and performance implications in a realistic setup.

7 Acknowledgments

This work was supported by Intel and University of Cyprus grants and in part by the European Commission under the Seventh Framework Programme through the SEARCHiN project (Marie Curie Action, contract number FP6-042467) and the Enabling Grids for E-sciencE project (contract number INFSO-RI-222667).

We would like to acknowledge the efforts of all Cyprus Grid members for providing us the support and resources for this work.

References

[1] Enabling Grids for E-sciencE. http://www.eu-egee.org/.

[2] ARM. Cortex-A8 Technical Reference Manual. 2007.

[3] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, Washington, DC, USA, 2004. IEEE Computer Society.

[4] A. Beszedes, R. Ferenc, T. Gyimuthy, A. Dolenc, and K. Karsisto. Survey of Code-Size Reduction Methods. *ACM Comput. Surv.*, 35(3), September 2003.

[5] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-execution: multicore caching for data-similar executions. In *ISCA*, June 2009.

[6] J. Casazza. First the tick, now the tock: Intel microarchitecture (nehalem). *Intel Corporation*.

[7] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. *SIGARCH Comput. Archit. News*, 33(2):357–368, 2005.

[8] K. D. Cooper and N. McIntosh. Enhanced Code Compression for Embedded RISC Processors. In *Proceedings of PLDI*, May 1999.

[9] S. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems*, 22(2), March 2000.

[10] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid - enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15:2001, 2001.

[11] S. Harizopoulos and A. Ailamaki. Improving instruction cache performance in oltp. *ACM Trans. Database Syst.*, 31(3):887–920, 2006.

[12] M. Kleanthous and Y. Sazeides. Catch: A mechanism for dynamically detecting cache-content-duplication and its application to instruction caches. In *DATE*, March 2008.

[13] D. Koufaty and D. T. Marr. Hyper-Threading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

[14] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving Code Density Using Compression Techniques. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 194–203, December 1997.

[15] C. Marco, C. Fabio, D. Alvise, C. Antonia, G. Francesco, M. Alessandro, M. Moreno, M. Salvatore, P. Fabrizio, P. Luca, and P. Francesco. The glite workload management system. In *4th International Conference on Grid and Pervasive Computing*, 2009.

[16] F. Mohamood, M. Ghosh, and H.-H. S. Lee. DLL-conscious Instruction Fetch Optimization for SMT Processors. *Journal of Systems Architecture*, 54:1089–1100, 2008.

[17] D. Sager, D. P. Group, and I. Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 2001.

[18] A. W. Services. Amazon elastic compute cloud: User guide. Technical Report API Version 2009-11-30, 2010.

[19] M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Saha, D. Sheahan, L. Spracklen, and A. Wynn. Ultrasparc t2: A highly-threaded, power-efficient, sparc soc. In *A-SSCC 2007*, November 2007.

[20] A. Shayesteh, G. Reinman, N. Jouppi, S. Sair, and T. Sherwood. Dynamically configurable shared cmp helper engines for improved performance. *SIGARCH Comput. Archit. News*, 33(4):70–79, 2005.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, October 2002.

[22] B. Sinharoy. Power7 multi-core processor design. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[23] A. J. Smith. Cache Memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, September 1982.

[24] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. *ACM SIGARCH Computer Architecture News*, 28(5):234–244, 2000.

[25] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[26] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Int. CMG Conference*, 1996.

[27] W. Yamamoto, M. Serrano, A. Talcott, R. Wood, and M. Nemirosky. Performance estimation of multistreamed, superscalar processors. In *Twenty-Seventh Hawaii Internation Conference on*, 1994.

A APPENDIX: Synthetic Benchmark to exercises Instruction Caches

```
void emptyFunc(){return;}
unsigned long long x = 0;

void oddN()     {
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        return;}
void evenN(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        return;}

void (*functionN[3])() = {&emptyFunc,&oddN,&evenN};

int execFlagN-1 = 1;
void oddN-1(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        execFlagN-1 = execFlagN-1 && !(depth == 2);
        int callFunc = gen_rand() & (execFlagN-1);
        functionN[execFlagN-1 + callFunc]();
        functionN[execFlagN-1 + ((callFunc^1) & execFlagN-1)]();
        execFlagN-1 ^= 1;
        return;}

void evenN-1(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        execFlagN-1 = execFlagN-1 && !(depth == 2);
        int callFunc = gen_rand() & (execFlagN-1);
        functionN[execFlagN-1 + callFunc]();
        functionN[execFlagN-1 + ((callFunc^1) & execFlagN-1)]();
        execFlagN-1 ^= 1;
        return;}

void (*functionN-1[3])() = {&emptyFunc,&oddN-1,&evenN-1};
.
.
.
void (*function2[3])() = {&emptyFunc,&odd2,&even2};

int execFlag1 = 1;
void odd1(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        execFlag1 = execFlag1 && !(depth == 1);
        int callFunc = gen_rand() & (execFlag1);
        function2[execFlag1 + callFunc]();
        function2[execFlag1 + ((callFunc^1) & execFlag1)]();
        execFlag1 ^= 1;
        return;}

void even1(){
        x = 0;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;
        execFlag1 = execFlag1 && !(depth == 1);
        int callFunc = gen_rand() & (execFlag1);
        function2[execFlag1 + callFunc]();
        function2[execFlag1 + ((callFunc^1) & execFlag1)]();
        execFlag1 ^= 1;
        return;}

void (*function1[3])() = {&emptyFunc,&odd1,&even1};

int main(int argc, char* argv[]){
        unsigned long long i = 0;
        unsigned long long k = atoi(argv[1]);
        depth = atoi(argv[2]);
        struct timeval t_start, t_fin;
        gettimeofday(&t_start,NULL);
        for (i = 0; i < k; i++){
        int callFunc = gen_rand() & (0x1);
        function1[callFunc+1]();
            function1[(callFunc ^ 1)+1]();}
        gettimeofday(&t_fin,NULL);
        timeval_subtract(&t_fin,&t_start);
        return 0;}
```